SCALING SEARCHABLE AND TRANSACTIONAL STORAGE SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Robert Charles Escriva

August 2017

SCALING SEARCHABLE AND

TRANSACTIONAL STORAGE SYSTEMS

Robert Charles Escriva, Ph.D.

Cornell University 2017

Data is the lifeblood of modern computing and the systems that store it have taken a prominent place in the infrastructure of practically every modern startup, business, or research application. Not-so-recent trends in distributed storage systems have removed features—such as secondary attribute search or transactions—that applications used to take for granted. These missing features must be reimplemented at the application level, or the application must be carefully constructed to work around their absence.

This thesis explores work on four systems that represent advances in reversing this trend. First, it looks at HyperDex, a system which provides efficient secondary attribute search. Second, it presents two transactional storage systems, Warp and Consus. Warp targets a single data center environment while Consus targets a geo-replicated deployment and the differences in their design reflect these two considerations. Finally, this thesis presents the Warp Transactional Filesystem that shows a positive example of how the transactional properties of Warp can be extended to provide application-level transactional guarantees. Finally, the thesis looks at the broader impact of these systems and how the evolution of the systems could be used to inform the development of future distributed systems.

BIOGRAPHICAL SKETCH

Robert Charles Escriva grew up in Macedon, NY with his parents Jill and Charlie, and his brother Carl. His interest in computing started at an early age where learning to navigate DOS was necessary to play the latest video games. Concurrently, his interest in ham radio put him in contact with many people who dabbled in and worked with radios and computers for a living. It is in this way that he came to own his first computer, a Commodore 64: In the third grade, an older ham gave him a well-loved C64; a TV that could not distinctly display `A`, `B`, `D`, `O`, `0`, or `8`; and a tape drive that looked like it originally played audio tracks. This single event most-directly lead Robert to discover programming and that the internals of a computer are much more like Legos® than black magic.

This interest in programming eventually led Robert to pursue a degree in Computer Science at Rensselaer Polytechnic Institute. At RPI, Robert became involved in undergraduate research opportunities after taking a course examining graph theory in the context of social networks. RPI also introduced Robert to working in the open source community with the Rensselaer Center for Open Source. His research collaborators and open source mentors were strong influences in his decision to go to grad school. It was these collaborations that also gave Robert an Erdős number of 2.

At Cornell, Robert discovered discovered a passion for systems research. Cornell's systems community taught Robert how to critically evaluate ideas and systems. Cornell also gave him the opportunity to build systems unencumbered by business considerations or other idea-limiting constraints; he was free to write code for the sake of intellectual satisfaction.

Robert will continue working in the field of systems and looks forward to finding opportunities for personal growth and systems research.

To my parents, family, and friends who made completing this journey possible.

Also, Hennessey for being there for me every day when I get home.

ACKNOWLEDGMENTS

Throughout my PhD career, I have encountered so many individuals willing to give me time and attention to help me become who I am today. Their time, assistance, and willingness to listen were indispensable and I do not believe this acknowledgements section will be sufficient to list everyone nor will it be sufficient to convey my gratitude.

First and foremost, I must thank those who formally advised me for teaching me what it means to do research, how to write a paper, and what it means to be a good member of the systems community. I want to thank Emin Gün Sirer for showing me how far I can push myself to do research. I want to thank Robbert van Renesse for helping me better understand the way I think about distributed systems, and how to map that onto the terminology, ideas, and thought processes of others. Finally, I want to thank my committee for investing time in supporting me through the formal steps of my PhD.

Throughout my PhD, I was lucky enough to interact with many great researchers at Cornell. I will deeply miss the systems lunch discussions we had every Friday, no matter how much of a detour they could take. I will miss the rivalry with the AI lunch at the same time slot. I would like to explicitly thank Fred B. Schneider for setting an example of what it means to be precise in one's language and understanding. I can only hope to some day approach this level of precision and understanding within a factor of two.

One of my favorite parts of Cornell has been and will always be the cordial nature of the department. I do not think there was a day that went by where passing someone in the hallway and saying hello would not elicit a warm smile and hello in response. In the snowy dark days of Ithaca, these smiles can be as uplifting as a warm cup of coffee. I would like to thank Dexter Kozen for al-

ways saying hello when we pass in the hallway and for his friendly and helpful advice.

Outside of Cornell, I'd like to thank the broader systems community. I regularly attended OSDI, SOSP, and NSDI and the people I met at these conferences will not be forgotten. I enjoyed the conversations at dinner and happy hour; I enjoyed the hallway discussions and debates. I hope that the submission of this thesis does not mark my departure from these communities.

Lastly, I would like to thank the friends and family who helped me navigate the tough decisions of grad school, who encouraged me to pursue my dreams, and who believed in me when I did not believe in myself. Your love and support over the last several years will be something I think about regularly whenever I encounter the tough obstacles of life.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgments . . . v
Table of Contents . . . vii
List of Tables . . . x
List of Figures . . . xi

1 Introduction **1**

2 Background **6**
2.1 Overview . . . 6
2.2 Key-Value Stores . . . 7
2.3 Search . . . 8
2.3.1 Single-Host Structures . . . 8
2.3.2 Peer-to-Peer Systems . . . 9
2.3.3 Space Filling Curves . . . 10
2.4 Transactions . . . 10
2.4.1 Concurrency Control . . . 11
2.4.2 Commit Protocols . . . 11
2.4.3 Consensus . . . 14
2.4.4 Synchronized Clocks . . . 15
2.4.5 Client Managed . . . 16
2.4.6 Static Analysis . . . 16
2.4.7 One Shot Transactions . . . 17
2.4.8 Geo-Replicated . . . 17
2.4.9 Filesystems . . . 18
2.5 Distributed Filesystems . . . 19

3 Secondary Attribute Search in HyperDex **21**
3.1 Introduction . . . 21
3.2 Approach . . . 23
3.2.1 Data Model and API . . . 24
3.2.2 Hyperspace Hashing . . . 24
3.2.3 Search Queries . . . 26
3.3 Data Partitioning . . . 28
3.3.1 Key Subspace . . . 29
3.3.2 Object Distribution Over Subspaces . . . 30
3.3.3 Heterogeneous Objects . . . 30
3.4 Consistency and Replication . . . 31
3.4.1 Value Dependent Chaining . . . 31
3.4.2 Fault Tolerance . . . 34
3.4.3 Consistency Guarantees . . . 35

3.5 Evaluation . . . 35
3.5.1 Get/Put Performance . . . 37
3.5.2 Search vs. Scan . . . 40
3.5.3 Scalability . . . 44
3.6 Conclusions . . . 46

4 Intra-Data Center Transactions in Warp **47**
4.1 Introduction . . . 47
4.2 Design . . . 49
4.2.1 Commit Protocol . . . 51
4.2.2 Fault Tolerance and Durability . . . 58
4.2.3 Atomicity, Consistency, Isolation . . . 59
4.2.4 Correctness . . . 59
4.3 Implementation . . . 61
4.3.1 Rich API . . . 62
4.3.2 Virtual Servers . . . 63
4.3.3 Coordinator . . . 64
4.4 Evaluation . . . 65
4.4.1 TPC-C Macro Benchmark . . . 66
4.5 Conclusion . . . 71

5 Geo-Replicated Transactions with Consus **72**
5.1 Introduction . . . 72
5.2 Design . . . 74
5.2.1 Commit Protocol . . . 76
5.2.2 Intra-Data Center Design . . . 83
5.3 Paxos Optimizations . . . 88
5.3.1 Avoiding Paxos . . . 88
5.3.2 Capturing Side Effects . . . 89
5.3.3 Implicit Phase 1 Paxos . . . 91
5.3.4 Recursive Generalized Paxos . . . 92
5.4 Implementation . . . 95
5.5 Evaluation . . . 96
5.6 Conclusion . . . 98

6 Transactional Filesystems with WTF **99**
6.1 Introduction . . . 99
6.2 Design . . . 101
6.2.1 The File Slicing Abstraction . . . 101
6.2.2 Storage Server Interface . . . 104
6.2.3 File Partitioning . . . 105
6.2.4 Filesystem Hierarchy . . . 107
6.2.5 File Slicing Interface . . . 108
6.2.6 Transaction Retry . . . 109

6.2.7 Locality-Aware Slice Placement 111
6.2.8 Metadata Compaction and Defragmentation 111
6.2.9 Garbage Collection 112
6.2.10 Fault Tolerance 113
6.3 Implementation 114
6.4 Evaluation 115
6.4.1 Applications 118
6.4.2 Micro Benchmarks 123
6.5 Conclusion 134

7 Conclusion **136**

Bibliography **140**

LIST OF TABLES

3.1 YCSB Workloads 38

4.1 TPC-C Workload Summary 67

5.1 An example where every transaction operation is learned by a quorum of the servers (3), but no operation is learned by all servers, and no server learns of all operations. 90

6.1 File-slicing API 106

6.2 File slicing reduces the amount of data written to the filesystem by one third compared to conventional APIs. 118

LIST OF FIGURES

3.1 Hyperspace hashing . . . 25
3.2 Subspace partitioning . . . 28
3.3 Value-dependent chaining . . . 33
3.4 YCSB Throughput Summary . . . 37
3.5 YCSB Worload A Latency CDF . . . 39
3.6 YCSB worload B Latency CDF . . . 40
3.7 YCSB Load Latency CDF . . . 41
3.8 YCSB Workload E Latency CDF . . . 42
3.9 Scan/Write Microbenchmark . . . 43
3.10 Subspace/Write Microbenchmark . . . 44
3.11 Scalabiity of HyperDex . . . 45

4.1 Warp system architecture . . . 50
4.2 Construction of transaction chains . . . 53
4.3 Detecting cyclic transactions . . . 56
4.4 TPC-C Throughput . . . 66
4.5 TPC-C New Order Latency . . . 67
4.6 TPC-C Payment Latency . . . 68
4.7 TPC-C Order Status Latency . . . 70

5.1 Consus system architecture . . . 75
5.2 Deadlock detection . . . 80
5.3 Transaction manager component . . . 84
5.4 Key-value store component . . . 86
5.5 TPC-C New Order Transactions . . . 97

6.1 WTF Architecture . . . 102
6.2 File-slicing abstraction . . . 103
6.3 File partitioning . . . 106
6.4 Sort benchmark . . . 115
6.5 Sort benchmark breakdown . . . 116
6.6 Work-queue application . . . 117
6.7 Video editor application . . . 120
6.8 WTF's transactional functionality enables users to manipulate the filesystem in isolation. . . . 122
6.9 Single-server baseline performance . . . 123
6.10 Throughput of sequential writers . . . 125
6.11 Latency of sequential writers . . . 126
6.12 Throughput of random writers . . . 127
6.13 Latency of random writers . . . 128
6.14 Throughput of sequential readers . . . 129
6.15 Throughput of random readers . . . 130
6.16 Write throughput with varying write load . . . 131

6.17 Write latency with varying write load 132
6.18 Throughput during failure and recovery 133
6.19 Garbage collection 134
6.20 Throughput of small sequential writes 135

CHAPTER 1

INTRODUCTION

Data is the lifeblood of modern computing and the systems that store it take a prominent place in the infrastructure of every modern startup, business, or research application. The storage systems have the challenging task of providing some degree of scalability, availability, and durability for the data. In an ideal—almost omniscient—system, the application would be free to access data as it chooses without having to structure itself around limitations in the storage system; data of interest to the application would always be accessible no matter how large the total data set and regardless of hardware or software failures. Such a system cannot be realized in practice and, instead, practitioners must balance numerous trade-offs between desirable properties.

One of the biggest differentiators in the design space is whether a system operates wholly within the confines of a single computer or is distributed across multiple computers. In general centralized systems are much easier to design and build; a distributed system inherits all the complexity and nuance of a single host system, multiplies it by the number of servers in the system, and adds additional overhead introduced by coordinating the components.

Not-so-recent trends in distributed storage systems improved performance or robustness by removing desirable features such as secondary attribute search or transactions. Without these features, the storage system can be much simpler and achieve higher performance. Applications that require these missing features must reimplement them at the application level or be designed to work around their absence. In both cases, the burden shifts from the storage system designer to the application.

This thesis describes four distributed storage systems that move functional-

ity back into the storage system while retaining the scalability, availability, and performance properties that make distributed systems appealing. A common strain through all of these systems is their deliberate construction around well-defined replication abstractions to provide more than just replication.

The first system, HyperDex, demonstrates how to extend a key-value store to have secondary attribute search using a technique called *hyperspace hashing*. Hyperspace hashing maps objects to servers by mapping both into a multi-dimensional hyperspace where searches naturally correspond to hyperplanes and geometric constructions in the hyperspace. HyperDex's replication protocol, called *value-dependent chaining*, chooses the replica set for an object based upon the current value of an object; the same object can have multiple distinct replica sets over its lifetime as its value changes.

The second system, Warp, extends the key-value interface of HyperDex to provide applications with cross-key transactions. The key insight in Warp is *transaction chaining*. Much like value-dependent chaining, transaction chaining constructs replica sets on the fly in order to replicate a transaction to all requisite servers. The construction ensures that transactions are serializable without any single server observing or ordering all transactions. The way Warp replicates transactions incurs the equivalent latency of one write operation per key in a transaction; consequently, write-heavy transactions can have latency that approaches that of a non-transactional workload while read-heavy transactions will incur a higher, but predictable, overhead compared to the same operations on a non-transactional store. For an implementation of the TPC-C benchmark, Warp achieves 75% of the throughput of the same workload when run against the non-transactional HyperDex interface.

The third system, Consus, achieves transactions equivalent to those enabled

by Warp in a geo-distributed system. A new commit protocol enables Consus to globally replicate and commit transactions in three message delays in the common case. Traditional two-phase commit requires a minimum of four message delays, while synchronously replicating non-transactional operations requires a minimum of two message delays. The Consus commit protocol embeds Generalized Paxos [66] to enable multiple distinct geographic sites to achieve consensus on the outcome of a transaction without having to serialize the decision through any one location. Consus uses one instance of Generalized Paxos per transaction to agree upon the disposition of the transaction at each data center. By limiting the scope of any given Generalized Paxos instance, Consus avoids many of the inefficiencies typically associated with consensus, leading to an efficient implementation.

The fourth system, the Warp Transactional Filesystem (WTF), provides a POSIX-like filesystem that allows groups of standard filesystem operations to be wrapped within serializable transactional blocks. The key insight in WTF is its division of internal state such that transactions over the entire system can be constructed using transactional operations over a much smaller amount of metadata. As the name implies the Warp Transactional Filesystem stores its metadata in Warp and derives all transactional guarantees from Warp itself; all code within WTF that deals with transactions does so using Warp.

The progression of these systems and their evolution provides insight not present in the individual presentation of each system [39, 40, 41, 42]. The evolution of these systems, and the dramatic changes in design between them provides a practical perspective of building distributed systems. While this thesis is not the first to address these practical considerations, addressing them provides context for the artifacts of the thesis and can inform future work in the area.

This thesis includes contributions that span multiple independent systems.

- This thesis introduces *hyperspace hashing*, a mapping technique for placing objects in a distributed system.
- This thesis introduces *value-dependent chaining*, a replication protocol that maintains an object across multiple independent mappings.
- This thesis evaluates HyperDex, a system implementing the principles of hyperspace hashing and value-dependent chaining.
- This thesis describes *linear transactions*, a transactions protocol that integrates replication with transaction commit to improve concurrency during transaction commit.
- This thesis evaluates Warp, a system built on HyperDex, implementing the principles of linear transactions.
- This thesis describes the commit protocol for Consus, a transactions protocol that can replicate and commit a transaction in the wide area in three message delays without a coordinator or distinguished proposer.
- This thesis evaluates Consus, a system implementing this commit protocol.
- This thesis introduces *file slicing*, an API that enables efficient file transformations such as reading and writing by reference.
- This thesis describes the design of a distributed transactional file system that supports the file slicing API.
- This thesis evaluates the Warp Transactional Filesystem, a system which implements the file slicing API and distributed transactions.

The rest of this thesis is organized as follows. Chapter 2 provides background on work related to this thesis. Chapter 3 describes secondary attribute

search in HyperDex. Chapter 4 describes the Warp extension to HyperDex. Chapter 5 describes geo-replicated transactions in Consus. Chapter 6 describes file slicing in the Warp Transactional Filesystem. Chapter 7 concludes the thesis with a discussion of the broader impact and applicability of this thesis.

CHAPTER 2

BACKGROUND

This chapter provides an overview of concepts related to the work presented in this thesis. It provides an overview of research on key-value stores, including many orthogonal or complementary research directions. Then it provides an overview of different related research on transactional systems. Finally, it provides an overview of distributed file systems.

2.1 Overview

Broadly speaking, the terms *key-value store* and *filesystem* are loosely defined: Both store data, both provide a means to access the data, and it is possible to construct a filesystem on top of a key-value store [35] or vice-versa [25]. Typically, a key-value store stores smaller objects—the Yahoo Cloud Serving Benchmark [30] specifies 1 kB—and provides primitives to read or write objects in their entirety. Advanced key-value stores may offer integrated read-modify-write operations, but the expectation is that objects are generally the unit of access. Key-value stores can either store the objects in a hash-table fashion, offering get/put primitives, or store the keys in lexicographically sorted order, offering a primitive to perform a range scan. A filesystem typically is built to store much larger objects (files) and provide the ability to efficiently read or update small parts of the larger object. Filesystems are hierarchical in nature and often provide the abstraction of files and directories. Complicating the terminology is the term *blob store*, which refers to something of a hybrid between a key-value store and a filesystem. A *blob store* typically refers to a system with a narrow read/write interface like a key-value store and the large-object capacity of a filesystem.

In the rest of this section, we will use the term key-value store to refer to storage systems meant to store and retrieve many small objects while filesystems refer to storage systems meant to store a smaller number of large objects.

2.2 Key-Value Stores

Modern key-value stores have their roots in work on distributed data structures [53] and distributed hash tables [60, 95, 97, 113, 127]. The work on distributed data structures outlines an architecture very much like a modern intra-data key-value store, where clients can talk to a key-value store node, which shares access to the hash table or lexicographically sorted map maintained within the system. Distributed hash tables (DHTs) provide a means of mapping objects to servers and routing scheme to access those objects. Key-value stores will often take advantage of the low latency and managed nature of data centers to employ a one-hop routing scheme rather than routing around a given geometry.

One of the most influential key-value stores is Dynamo [37], Amazon's in-house key-value store originally designed to maintain shopping carts. Dynamo, in turn, uses a consistent hashing ring [60, 113] to map objects to servers and a sloppy-quorum system to maintain write availability in the face of even the most extreme failure scenarios. Popular open-source systems like Voldemort [93] or Riak [96] are heavily influenced by Dynamo's design.

Another influential key-value store is Google's BigTable [25], which maintains a lexicographically sorted multi-map. BigTable builds on Google's GFS [47] and uses the fault tolerance of the filesystem to provide fault tolerance for the key-value store. The system partitions the data set into *tablets* and maintains a 3-level hierarchy of tablets to provide fast and consistent access to data. Popular open-source systems like HBase [12] and HyperTable [59] are heavily influenced

by the design of BigTable.

Other systems employ different designs to present something akin to a key-value interface. For example, Yahoo!'s PNUTS [29] system supports selection and projection functions but does not support joins and Cassandra [65] is influenced by BigTable's API and Dynamo's ring structure.

Finally, there are many strains of research that look at improving performance of key-value storage. Fawn KV [7] builds a key-value store on underpowered hardware to improve the throughput-to-power-draw ratio. SILT [71] eliminates read amplification to maximize read bandwidth in a datastore backed by solid-state disk. RAMCloud [89] stores data in RAM and utilizes fast network connections to rapidly restore failed replicas. TSSL [112] utilizes a multi-tier storage hierarchy to exploit cache-oblivious algorithms in the storage layer. Masstree [78] uses concatenated B^+ trees to service millions of queries per second. FaRM [38] uses RDMA to build a fast in-memory key-value store with single-machine transactions.

2.3 Search

2.3.1 Single-Host Structures

Storage systems that organize their data in high-dimensional spaces were pioneered by the database community more than thirty years ago [16, 19, 54, 62, 86, 90, 98]. These systems, collectively known as Multi-Dimensional Databases, leverage multi-dimensional data structures to improve the performance of data warehousing and online analytical processing applications.

The key aspect of each of these indexing techniques is that they create and maintain a structure for multi-dimensional indexing. By design, they operate

on a single machine, where the invariants of the structure can be easily maintained. Because the index is updated with each new object, the data structure continuously changes in write heavy workloads and in a distributed setting, it is not readily apparent how to map any given structure onto multiple hosts. There are systems that explore how to efficiently maintain large-scale tree-based distributed data structures by directly building single-host data structures in a distributed setting [3, 34].

2.3.2 Peer-to-Peer Systems

Work in peer-to-peer systems has explored multi-dimensional network overlays to facilitate decentralized data storage and retrieval. MURK [45], SkipIndex [125], and SWAM-V [61] dynamically partition the multi-dimensional space into kd-trees, skip graphs, and Voronoi diagrams respectively to provide multi-dimensional range lookups. Each of these systems builds and maintains a structure for indexing; as the data grows, the system will have an increasing cost to manage the structure compared to the cost of inserting each individual data item.

Other approaches construct a multi-dimensional index on top of a single-dimensional indexing system. Mercury [21] builds on top of a Chord [113] ring, and uses consistent hashing [60] on each attribute as secondary indexes. A search in Mercury essentially joins these rings to return results. Arpeggio[28] provides search over multiple attributes by enumerating and creating an index of all $\binom{k}{x}$ fixed-size subsets of attributes using a Chord ring. Both of these approaches insert redundant pointers into rings.

2.3.3 Space Filling Curves

A common approach to providing multi-attribute search uses space-filling curves to map multi-dimensional data into a single dimension, and then partition this mapping across a set of hosts. SCRAP [45], Squid [101] and ZNet [108] are examples of this approach with each node responsible for data in a contiguous range of the space filling curve. Similarly, MAAN [24] performs the same mapping, but uses uniform locality preserving hashing.

Space-filling curves and other techniques can be suboptimal when dimensionality is high because a single query may intersect the curve in many places. A good choice of space filling curve will ensure that points which are close together in the multi-dimensional space are close to each other on the curve; however, for some points in the space, objects close to each other will be quite far from each other on the curve. A range query spanning such points will necessarily cover multiple distinct segments of the curve. For queries that specify a subset of the attributes used to construct the curve, there could be many distinct ranges of the curve that must be queried to return results.

2.4 Transactions

Transaction management is a complex and nuanced subject. Systems are often presented holistically and features or contributions that could be cleanly separated in one design may be enmeshed in other designs; terminology or classifications that apply to one system may not cleanly apply to other systems. Broadly speaking, this section presents different work in the space of transactions by grouping similar techniques, even though one system may have contributions that span many or all of the different categories.

2.4.1 Concurrency Control

Concurrency control restricts the interaction of two or more transactions in order to uphold system-wide guarantees. Broadly speaking, concurrency control traditionally has been described as *pessimistic* when a system mechanism actively prevents two concurrent transactions from interacting with each other and *optimistic* when transactions' executions are largely unimpeded and a mechanism checks for correct concurrent behavior before commit.

Traditional pessimistic transaction managers perform lock or timestamp management [20], and employ a separate protocol for coordination between hosts. One characteristic of such systems is that when two concurrent transactions interact, one will wait for the other to commit.

In optimistic concurrency control, transactions will run to completion and employ a validation step. When two optimistic transactions conflict, one will abort to allow the other to proceed. In general, optimistic schemes can be divided into backward-oriented and forward-oriented schemes [118]. In a backward-oriented scheme, reads are checked against previously performed writes; essentially, the pending transaction is compared to the history of the system to see if it can still commit. Forward-oriented optimistic concurrency control checks a committing transaction's writes against concurrently executing, unvalidated reads.

2.4.2 Commit Protocols

Commit protocols serve to coordinate a transaction's commit across multiple servers. A commit protocol typically ensures that transactions commit atomically—that is either the entire transaction commits or none of it does.

Two-phase commit [51] is a classic commit protocol. Under 2PC, a single

server called a coordinator tries to apply a transaction to a set of participants. In the first phase, the coordinator asks every participant to prepare the transaction. The prepare step asks each participant to state whether the transaction can commit at the participant, and uphold this answer until the protocol completes. A transaction may commit if and only if all participants can commit. In the second phase, the coordinator decides if it can commit the transaction and informs all participants.

The limitations of the two-phase commit protocol become apparent when the coordinator or participants may fail. Because every server must be alive to positively signal the transaction may commit, a single failure can prevent a transaction from committing. Further, if the coordinator fails, the system as a whole is left in an undetermined state. In general, 2PC makes no assumptions about the network, but does assume that the participants and the coordinator will not fail. It is possible to amend the protocol to support recovery from failure of a single participant or the coordinator; the failure of a participant and the coordinator cannot easily be recovered from. These amended variants often occur in practice where the coordinator can fail and a new coordinator can pickup the transaction and resume. They assume that the coordinator can be reliably replaced and that no participants fail during this process. Such variations are rigid in the failure patterns they can overcome. Multiple failures or repeated failures are generally problematic and require the system to make additional assumptions.

Three-phase commit addresses these failure cases by introducing some synchrony assumptions. While classic 2PC makes no assumptions about network delivery, the speed of individual nodes, or relative processing speed of the nodes, 3PC assumes fail-stop [100] where node failures can be reliably detected.

This difference allows the coordinator and participants to mutually detect each others' failure. On failure, each step of the protocol has a well-defined fallback to abort or commit.

The tension between 2PC and 3PC illustrates two ends of a spectrum for assumptions. 2PC makes no synchrony assumptions, and consequently does not tolerate failure in its most basic form. 3PC makes very strong synchrony assumptions in order to always overcome failure, and its correctness relies upon the extent to which these assumptions hold in practice.

Gray and Lamport [52] show how to use Paxos to create a transaction commit protocol, where each participant uses Paxos to record its decision to commit or abort. The "coordinator" for this protocol, then, exists in distributed quorums that include a subset of the participants. In general, 2PC, 3PC, and derivatives are comparable to a special $f = 0$ case of Paxos Commit that cannot tolerate coordinator failure. This Paxos-derived commit protocol is robust in the face of failures—it can tolerate a minority of server failures—and makes no more assumptions about the network than 2PC.

There is an interesting cross-over between commit protocols like 2PC and 3PC and concurrency control. Because the commit protocol specifies that a participant who votes to `commit` must ensure the transaction can commit in the second phase, the commit protocol effectively places requirements upon the concurrency control to uphold this invariant. For example, Sinfonia [4] introduces the *mini-transaction* primitive which combines 2PC with optimistic concurrency control. In the first phase, the protocol performs backward-oriented optimistic concurrency control in addition to preparing the transaction. In the second phase, the transaction commits as in 2PC.

Sinfonia's design highlights the interaction between optimistic concurrency

control and commit protocols. Even in an optimistic scheme, the concurrency control generally has a period of time during which the commit protocol forces transactions to act pessimistically. Similarly, Google's Spanner [31] uses 2PC, but buffers writes and acquiring the accompanying write lock until commit time, and the write locks are acquired during phase one of commit. While Spanner is self-described as using pessimistic locking, the time period during which writes are visible to concurrency control strictly overlaps with the commit protocol. These cases of Spanner and Sinfonia point to a potential for future work to rethink the distinction between optimistic and pessimistic concurrency control and the separation of concurrency control from the commit protocol.

2.4.3 Consensus

Consensus protocols provide a means of agreeing upon values across multiple machines. Such agreement has a natural place in designing transactional systems, and recent work has examined how to use a general consensus protocol, such as Paxos [67] or Zab [58], to serialize transactions in a fault-tolerant manner.

The Calvin [120] system uses ZooKeeper/Zab to order transactions as input to the system, and then processes them in the order dictated by ZooKeeper. Consequently, all transactions are serialized in a particular order and all servers apply transactions in the specified order. A technique for deterministically executing a sequence of transactions allows the system to achieve parallelism without requiring all servers to move in lock step at the speed of a single server.

Another approach to using consensus for transactions is to create independent partitions of data, where each partition runs its own consensus algorithm [14, 50, 99, 105]. In this technique, transactions that can be contained entirely within

a single partition commit and execute just as in Calvin. Transactions spanning multiple partitions are either not supported, or use a commit protocol like 2PC to commit the transaction atomically on two partitions.

Orthogonal to data partitioning is the idea of transaction batching [104, 120]. Transaction batching assumes that consensus is expensive and that throughput can be increased by grouping multiple transactions into the same round of consensus to improve throughput. Thus, one round of consensus decides a set of transactions potentially multiplying the throughput of the system by a factor correlated with batch size.

Finally, techniques based upon Generalized Paxos [66] allow the system to commit in parallel transactions that do not interact. MDCC [63] uses Generalized Paxos and a demarcation protocol to allow transactions to extract more parallelism. Related to Generalized Paxos is Egalitarian Paxos [83], which provides high performance Paxos in the wide area. Egalitarian Paxos could be used as the underpinnings of a system like Calvin or any other non-general consensus protocol.

2.4.4 Synchronized Clocks

Synchronized clocks can make transaction management easier: A synchronized clock provides different nodes in a distributed system with the ability to infer information about the state of other nodes. In systems like the Thor project [2], clocks are loosely synchronized and allow nodes to cache data for transactional access; a node can use the time to rule out certain states and thus reduce its overhead. Similarly, Granola [33] orders independent transactions with no locking overhead or abort mechanism, and orders these transactions using time synchronization as an optimization.

With many transactional systems that rely upon clock synchronization, the synchronization acts as an optimization. As systems become less synchronized, the effects of the optimization can become less pronounced, but the correctness of the system is preserved. Another point in this design space is to establish tight bounds on clock skew and drift. Spanner [32] uses a technique called TrueTime that provides an interval within which the "real" time exists. Servers can safely use the center of this interval as the true time, and then use the size of the interval to wait out any uncertainty in order to ensure that all clocks pass a certain time. In Spanner, a bad clock will allow the system to break its guarantee to the client; however, the probability of such a result is carefully controlled in a managed environment.

2.4.5 Client Managed

Some systems build on existing storage by implementing transactions directly inside a client library. Such systems use the unmodified API of a non-transactional system and present a transactional API to the application. CrSO [43] provides snapshot isolation using HBase's multi-version storage and a centralized status oracle to check for read-write or write-write conflicts at commit time. Percolator [91] provides a transactional interface on top of BigTable by storing per-item locks and transactional metadata within BigTable itself.

2.4.6 Static Analysis

Static analysis is useful for determining a priori if a transaction workload can be decomposed to run in a distributed fashion while maintaining serializability. Lynx [126] decomposes transactions into multiple atomic units and proves

that, so long as each unit is atomic, transactions' units may be allowed to interleave without loss of guarantees. Rococo [84] also requires offline analysis of transactions in order to decompose them into smaller atomic units, which it then executes in parallel. Both of these systems use static analysis to provide serializable transactions.

Recent work on $\mathcal{I}$-confluence analysis [13] explores using static analysis to specialize the guarantees provided by a database to uphold a programmer-specified set of invariants and minimize coordination.

2.4.7 One Shot Transactions

H-Store [114] introduced the idea of a *one-shot transaction*. A one-shot transaction is a transaction that can execute on each participant or partition independent from the transactions on another partition; the data dependencies on one partition are satisfiable entirely within that partition. The restricted nature of a one-shot transaction allows concurrency control and the commit protocol to be merged into one [85].

2.4.8 Geo-Replicated

Some transactional systems are specifically designed for use in a geo-replicated setting. Spanner [31] uses TrueTime to provide world-wide external consistency. Replicated Commit [77] flips the layering of 2PC and Paxos from what it is in Spanner such that each data center runs its own instance of 2PC and uses Paxos to reach consensus across the 2PC groups. By switching the layering of 2PC and Paxos, Replicated commit is able to alter the number of cross-data center round trips necessary to execute a transaction. Whether Spanner or Replicated

Commit yield a lower latency is largely a result of the number of data centers and the number of reads within a transaction.

Other geo-replicated systems often accept weaker consistency guarantees in order to ensure lower latency. COPS-GT [74] and Eiger [75] provide read and write transactions, respectively, that commit locally and propagate to remote data centers in a causally-consistent fashion. Walter [110] implements parallel snapshot isolation using counting sets to resolve conflicting versions, similar to commutative data types [70]. Gemini uses a unique red-blue labeling for operations to mix consistency in the wide area.

2.4.9 Filesystems

Transactional filesystems enable applications to offload much of the hard work relating to update consistency and durability to the filesystem. The QuickSilver operating system shows that transactions across the filesystem simplify application development [103]. Further work showed that transactions could be easily added to LFS, exploiting properties of the already-log-structured data to simplify the design [107]. Valor [111] builds transaction support into the Linux kernel by interposing a lock manager between the kernel's VFS calls and existing VFS implementations.

Optimistic concurrency control schemes often enable more concurrency for lightly-contended workloads. PerDiS FS adopts an optimistic concurrency control scheme that relies upon external actions to reconcile concurrent changes to a file [46]. This allows users and applications to concurrently work on the same file. Liskov and Rodrigues show that much of the overhead of a serializable filesystem can be avoided by running read-only transactions in the recent past, and employing an optimistic protocol for read-write transactions [73].

This thesis is not the first to build a transactional filesystem on top of a transactional data store. Inversion [88] builds on PostgreSQL to maintain a complete filesystem. KBDBFS [111] and Amino [124] both build on top of BerkeleyDB; the former is an in-kernel implementation of BerkeleyDB, while the latter eschews the complexity and takes a performance hit with a userspace implementation.

2.5 Distributed Filesystems

Distributed filesystems expose one or more units of storage over a network to clients. AFS [57] exports a uniform namespace to workstations, and stores all data on centralized servers. Other systems [55, 102, 106], most notably xFS [8] and Swift [23] stripe data across multiple servers for higher performance than can be achieved with a single disk. Petal [69] provides a virtual disk abstraction that clients may use as a traditional block device. Frangipani [117] builds a filesystem abstraction on top of Petal. NASD [48] and Panasas [123] employ customized storage devices that attach to the network to store the bulk of the metadata.

Farsite [1] separates data from metadata to implement a byzantine fault tolerant filesystem where only the metadata replicas employ BFT algorithms. The insight in Farsite is that, while BFT will not scale to the size of a filesystem, it is sufficient to enable BFT guarantees on the metadata replicas to extend some BFT guarantees across the entire filesystem.

Recent work focuses on building large-scale data-center-centric filesystems. GFS [47] and HDFS [9] employ a centralized server that maintains the metadata, mediates client access, and coordinates the storage servers. Salus [122] improves HDFS to support storage and computation failures, but retains the central metadata server. This centralized master approach, however, suffers from scalability

bottlenecks inherent to the limits of a single server [79].

CalvinFS [119] provides a transactional filesystem by storing and manipulating metadata using Calvin [120]. Transactions in CalvinFS are limited, and cannot do read-modify-write operations on the filesystem without additional mechanism. Further, CalvinFS addresses file fragmentation using a garbage collection mechanism that entirely rewrites fragmented files; in the worst case, a sequential writer could incur I/O that scales quadratically in the size of the file.

Another approach to scalability is demonstrated by Flat Datacenter Storage [87], which enables applications to access any disk in a cluster via a CLOS network with full bisection bandwidth. To eliminate the scalability bottlenecks inherent to a single master design, FDS stores metadata on its tract servers and uses a centralized master solely to maintain the list of servers in the system. Blizzard [81] builds block storage, visible to applications as a standard block device, on top of FDS, using nested striping and eventual durability to service the smaller writes typical of POSIX applications.

Blob storage systems behave similarly to filesystems, but with a restricted interface that permits creating, retrieving, and deleting blobs, without efficient support for arbitrarily changing or resizing blobs. Facebook's f4 [115] ensures infrequently accessed files are readily available. Pelican [15] enables power-efficient cold storage by over provisioning storage, and selectively turning on subsets of disks to service requests. The design goals of these systems are provide a subset of the operations used by applications built for filesystems; filesystems could easily be used to generate, maintain, and modify data before placing it into blob storage.

CHAPTER 3

SECONDARY ATTRIBUTE SEARCH IN HYPERDEX

3.1 Introduction

Prominent distributed key-value stores such as BigTable [25], Cassandra [65] and Dynamo [37] form the backbone of large commercial applications because they offer scalability and availability properties that traditional database systems do not provide. Yet these properties come at a substantial cost: the data retrieval API is often narrow and restrictive, requiring systems implement costly secondary indexing schemes or enumerate all objects of a given type in order to retrieve objects using any method other than retrieval by primary key.

This chapter introduces HyperDex, a high-performance, scalable, consistent and distributed key-value store that provides a new `search` primitive for retrieving objects by secondary attributes. HyperDex achieves this extended functionality by organizing its data using a novel technique called *hyperspace hashing*. Similar to other hashing techniques [53, 60, 80, 95], hyperspace hashing deterministically maps objects to servers to enable efficient object insertion and retrieval. But it differs from these techniques because it takes into account the secondary attributes of an object when determining the mapping for an object. Specifically, it maps objects to coordinates in a multi-dimensional Euclidean space—a *hyperspace*—which has axes defined by the objects' attributes. Each server in the system is mapped onto a region of the same hyperspace, and owns the objects that fall within its region. Clients use this mapping to deterministically insert, remove, and search for objects.

Hyperspace hashing facilitates efficient search by significantly reducing the number of servers to contact for each partially-specified `search`. The construc-

tion of the hyperspace mapping guarantees that objects with matching attribute values will reside on the same server. Through geometric reasoning, clients can restrict the search space for a partially-specified query to a subset of servers in the system, thereby improving search efficiency. Specificity in searches works to the clients' advantage: a fully-specified search contacts exactly one server.

A naive hyperspace construction, however, may suffer from a well-known problem with multi-attribute data known as "curse of dimensionality [18]." With each additional secondary attribute, the hyperspace increases in volume exponentially. If constructed in this fashion, each server would be responsible for a large volume of the resulting hyperspace, which would in turn force search operations to contact a large number of servers, counteracting the benefits of hyperspace hashing. HyperDex addresses this problem by partitioning the data into smaller, limited size *subspaces* of fewer dimensions.

Failures are inevitable in large-scale deployments. Standard approaches for providing fault tolerance store objects on a fixed set of replicas determined by a primary key. These techniques, whether they employ a consensus algorithm among the replicas and provide strong consistency [31, 94], or spray the updates to the replicas and only achieve eventual consistency [37, 65, 93, 96], assume that the replica sets remain fixed even as the objects are updated. Such techniques are not immediately suitable in this setting because, in hyperspace hashing, object attributes determine the set of servers on which an object resides, and consequently, each update may implicitly alter the replica set. Providing strong consistency guarantees with low overhead is difficult, and more so when replica sets change dynamically and frequently. HyperDex utilizes a novel replication protocol called *value-dependent chaining* to simultaneously achieve fault tolerance, high performance and strong consistency. Value-dependent chaining

replicates an object to withstand f faults (which may span server crashes and network partitions) and ensures linearizability, even as replica sets are updated. Thus, HyperDex's replication protocol guarantees that all `get` operations will immediately see the result of the last completed `put` operation—a stronger consistency guarantee than those offered by the current generation of NoSQL data stores.

Overall, this chapter describes the architecture of a key-value store whose API is one step closer to that of traditional RDBMSs while offering strong consistency guarantees, fault-tolerance for failures affecting a threshold of servers, and high performance, and makes three contributions. First, it describes a new hashing technique for mapping structured data to servers. This hashing technique enables efficient retrieval of multiple objects. Second, it describes a fault-tolerant, strongly-consistent replication scheme that accommodates object relocation. Finally, it reports from a full implementation of the system and deployment in a data center setting consisting of 64 servers, and demonstrates that HyperDex provides performance that is comparable to or better than Cassandra and MongoDB, two current state-of-the-art cloud storage systems, as measured using the industry-standard YCSB [30] benchmark. More specifically, HyperDex achieves 12-13$\times$ higher throughput for search workloads than the other systems, and consistently achieves 2-4$\times$ higher throughput for traditional key-value workloads.

3.2 Approach

This section describes the data model used in HyperDex, outlines hyperspace hashing, and sketches the high-level organization and operation of the system.

3.2.1 Data Model and API

HyperDex stores objects that consist of a key and zero or more secondary attributes. As in a relational database, HyperDex objects match an application-provided schema that defines the typed attributes of the object and are persisted in tables. This organization permits straightforward migration from existing key-value stores and database systems.

HyperDex provides a rich API that supports a variety of data structures and a wide range of operations. The system natively supports primitive types, such as `string`s, `integer`s and `float`s, as well as composite types, such as `list`s, `set`s or `map`s constructed from primitive types. The dozens of operations that HyperDex provides on these data types fall into three categories. First, *basic operations*, consisting of `get`, `put`, and `delete`, enable a user to retrieve, update, and destroy an object identified by its key. Second, the `search` operation enables a user to specify zero or more ranges for secondary attributes and retrieve the objects whose attributes fall within the specified, potentially singleton, ranges. Finally, atomic operations, such as `compare-and-swap` and `atomic-inc`, enable applications to safely perform concurrent updates on objects identified by their keys. The composite types and atomic operations provided by HyperDex are built on top of the more basic `get`, `put`, `delete`, and `search` operations that form the core of HyperDex. Consequently this chapter focuses on the basic operations.

3.2.2 Hyperspace Hashing

HyperDex represents each table as an independent multi-dimensional space, where the dimensional axes correspond directly to the attributes of the table. HyperDex assigns every object a corresponding coordinate based on the object's

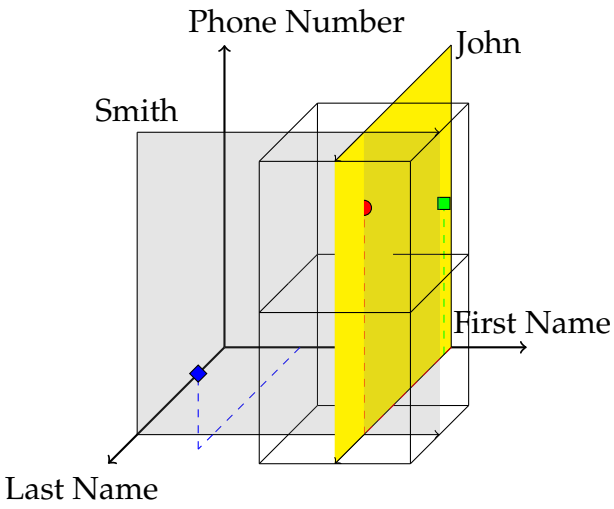


Figure 3.1: Simple hyperspace hashing in three dimensions. Each plane represents a query on a single attribute. The plane orthogonal to the axis for "Last Name" passes through all points for `last_name = 'Smith'`, while the other plane passes through all points for `first_name = 'John'`. Together they represent a line formed by the intersection of the two search conditions; that is, all phone numbers for people named "John Smith". The two cubes show regions of the space assigned to two different servers. The query for "John Smith" needs to contact only these servers.

attribute values. An object is mapped to a deterministic coordinate in space by hashing each of its attribute values to a location along the corresponding axis.

Consider, for the following discussion, a table containing user information that has the attributes `first-name`, `last-name`, and `telephone-number`. For this schema, HyperDex would create a three dimensional space where the first-name attribute comprises the x-axis, the last-name attribute comprises the y-axis, and the telephone-number attribute comprises the z-axis. Figure 3.1 illustrates this mapping.

Hyperspace hashing determines the object to server mapping by tessellating the hyperspace into a grid of non-overlapping regions. Each server is assigned, and is responsible for, a specific region. Objects whose coordinates fall within a region are stored on the corresponding server. Thus, the hyperspace tessalation

serves like a multi-dimensional hash bucket, mapping each object to a unique server.

The tessalation of the hyperspace into regions (called the *hyperspace mapping*), as well as the assignment of the regions to servers, is performed by a fault-tolerant coordinator. The primary function of the coordinator is to maintain the hyperspace mapping and to disseminate it to both servers and clients. The hyperspace mapping is initially created by dividing the hyperspace into hyperrectangular regions and assigning each region to a virtual server. The coordinator is then responsible for maintaining this mapping as servers fail and new servers are introduced into the system.

The geometric properties of the hyperspace make object insertion and deletion simple. To insert an object, a client computes the coordinate for the object by hashing each of the object's attributes, uses the hyperspace mapping to determine the region in which the object lies, and contacts that server to store the object. The hyperspace mapping obviates the need for server-to-server routing.

3.2.3 Search Queries

The hyperspace mapping described in the preceding sections facilitates a geometric approach to resolving `search` operations. In HyperDex, a `search` specifies a set of attributes and the values that they must match (or, in the case of numeric values, a range they must fall between). HyperDex returns objects which match the `search`. Each `search` operation uniquely maps to a *hyperplane* in the hyperspace mapping. A `search` with one attribute specified maps to a hyperplane that intersects that attribute's axis in exactly one location and intersects all other axes at every point. Alternatively, a `search` that specifies all attributes maps to exactly one point in hyperspace. The hyperspace map-

ping ensures that each additional search term potentially reduces the number of servers to contact and guarantees that additional search terms will not increase `search` complexity.

Clients maintain a copy of the hyperspace mapping, and use it to deterministically execute `search` operations. A client first maps the `search` into the hyperspace using the mapping. It then determines which servers' regions intersect the resulting hyperplane, and issues the `search` request to only those servers. The client may then collect matching results from the servers. Because the hyperspace mapping maps objects and servers into the same hyperspace, it is never necessary to contact any server whose region does not intersect the `search` hyperplane.

Range queries correspond to extruded hyperplanes. When an attribute of a `search` specifies a range of values, the corresponding hyperplane will intersect the attribute's axis at every point that falls between the lower and upper bounds of the range. Note that for such a scheme to work, objects' relative orders for the attribute must be preserved when mapped onto the hyperspace axis.

Figure 3.1 illustrates a query for `first_name = 'John'` and `last_name = 'Smith'`. The query for `first_name = 'John'` corresponds to a two-dimensional plane which intercepts the `first_name` axis at the hash of `'John'`. Similarly, the query for `last_name = 'Smith'` creates another plane which intersects the `last_name` axis. The intersection of the two planes is the line along which all phone numbers for John Smith reside. Since a search for John Smith in a particular area code defines a line segment, a HyperDex search needs to contact only those nodes whose regions intersect that segment.

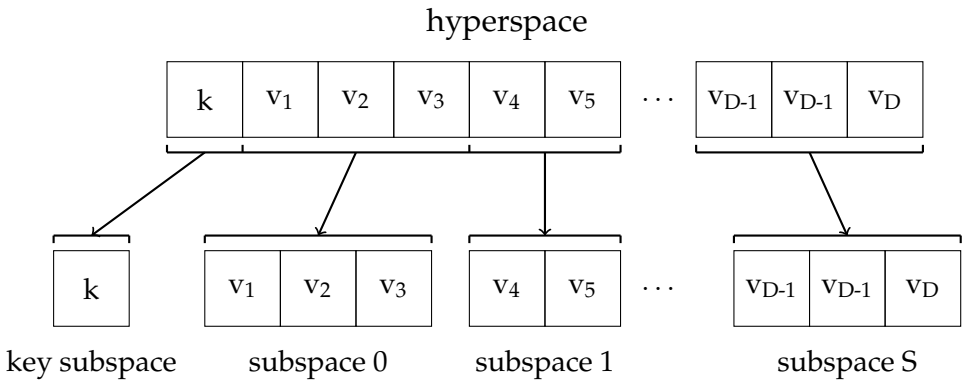


Figure 3.2: HyperDex partitions a high-dimensional hyperspace into multiple low-dimension subspaces.

3.3 Data Partitioning

HyperDex's Euclidean space construction significantly reduces the set of servers that must be contacted to find matching objects.

However, the drawback of coupling the dimensionality of hyperspace with the number of searchable attributes is that, for tables with many searchable attributes, the hyperspace can be very large since its volume grows exponentially with the number of dimensions. Covering large spaces with a grid of servers may not be feasible even for large data-center deployments. In general, a unpartitioned D dimensional hyperspace will need $O(2^D)$ servers.

HyperDex avoids the problems associated with high-dimensionality by partitioning tables with many attributes into multiple lower-dimensional hyperspaces called *subspaces*. Each of these subspaces uses a subset of object attributes as the dimensional axes for an independent hyperspace. Figure 3.2 shows how HyperDex can partition a table with D attributes into multiple independent subspaces. When performing a `search` on a table, clients select the subspace that contacts the fewest servers, and will issue the `search` to servers in exactly one subspace.

Data partitioning can increase the efficiency of a `search` by reducing the dimensionality of the underlying hyperspace. The trade-off partitioning makes is that searching may involve consulting the mapping of multiple hyperspaces of low-dimensionality instead of a single hyperspace with many dimensions. By partitioning the table, HyperDex reduces the worst case behavior, decreases the number of servers necessary to maintain a table, and decreases the number of servers a `search` is likely to contact.

Data partitioning forces a trade-off between search generality and efficiency. On the one hand, a single hyperspace can accommodate arbitrary searches over its associated attributes. On the other hand, a hyperspace which is too large will always require that partially-specified queries contact many servers. Since applications often exhibit search locality, HyperDex applications can tune search efficiency by creating corresponding subspaces. As the number of subspaces grows, so, too, do the costs associated with maintaining data consistency across subspaces. Section 3.4 details how HyperDex efficiently maintains consistency across subspaces while maintaining a predictably low overhead.

3.3.1 Key Subspace

The basic hyperspace mapping, as described so far, does not distinguish the key of an object from its secondary attributes. This leads to two significant problems when implementing a practical key-value store. First, key lookups would be equivalent to single attribute searches. Although HyperDex provides efficient search, a single attribute search in a multi-dimensional space would likely involve contacting more than one server. In this hypothetical scenario, key operations would be strictly more costly than key operations in traditional key-value stores.

HyperDex provides efficient key-based operations by creating a one-dimensional subspace dedicated to the key. This subspace, called the *key subspace*, ensures that each object will map to exactly one region in the resulting hyperspace. Further, this region will not change as the object changes because keys are immutable. To maintain the uniqueness invariant, `put` operations are applied to the key subspace before the remaining subspaces.

3.3.2 Object Distribution Over Subspaces

Subspace partitioning exposes a design choice in how objects are distributed and stored on servers. One possible design choice is to keep data in normalized form, where every subspace retains, for each object, only those object attributes that serve as the subspace's dimensional axes. While this approach would minimize storage requirements per server, as attributes are not duplicated across subspaces, it would lead to more expensive search and object retrieval operations since reconstituting the object requires cross-server cooperation. In contrast, an alternative design choice is to store a full copy of each object in each subspace, which leads to faster search and retrieval operations at the expense of additional storage requirements per server.

Hyperspace hashing supports both of these object distribution techniques. The HyperDex implementation, however, relies upon the latter approach to implement the replication scheme described in Section 3.4.

3.3.3 Heterogeneous Objects

In a real deployment, the key-value store will likely be used to hold disparate objects with different schema. HyperDex supports this through the table ab-

straction. Each table has a separate set of attributes which make up the objects within, and these attributes are partitioned into subspaces independent of all other tables. As a result, HyperDex manages multiple independent hyperspaces.

3.4 Consistency and Replication

Because hyperspace hashing maps each object to multiple servers, maintaining a consistent view of objects poses a challenge. HyperDex employs a novel technique called *value-dependent chaining* to provide strong consistency and fault tolerance in the presence of concurrent updates.

For clarity, this section first presents value-dependent chaining without concern for fault tolerance. Under this scheme, a single failure leaves portions of the hyperspace unavailable for updates and searches. The chapter then describes how to extend value-dependent chaining such that the system can tolerate up to f failures in any one region.

3.4.1 Value Dependent Chaining

Because hyperspace hashing determines the location of an object by its contents, and subspace partitioning creates many object replicas, objects will be mapped to multiple servers and these servers will change as the objects are updated. Change in an object's location would cause problems if implemented naively. For example, if object updates were to be implemented by simply sending the object to all affected servers, there would be no guarantees associated with subsequent operations on that object. Such a scheme would at best provide eventual consistency because servers may receive updates out-of-order, with no sen-

sible means of resolving concurrent updates.

HyperDex orders updates by arranging an object's replicas into a value-dependent chain whose members are deterministically chosen based upon an object's hyperspace coordinate. The head of the chain is called the *point leader*, and is determined by hashing the object's key. Subsequent servers in the chain are determined by hashing attribute values for each of the remaining subspaces.

This construction of value-dependent chains enables efficient, deterministic propagation of updates. The point leader for an object is in a position to dictate the total order on all updates to that object. Each update flows from the point leader through the chain, and remains pending until an acknowledgement of that update is received from the next server in the chain. When an update reaches the tail, the tail sends an acknowledgement back through the chain in reverse so that all other servers may commit the pending update and clean up transient state. When the acknowledgement reaches the point leader, the client is notified that the operation is complete. Figure 3.3, illustrates a value-dependent chain across three subspaces.

Updates to preexisting objects are more complicated because a change in an attribute value might require relocating an object to a different region of the hyperspace. Value-dependent chains address this by incorporating the servers assigned to regions for both the old and new versions of the object. Chains are constructed such that servers are ordered by subspace and the servers corresponding to the old version of the object immediately precede the servers corresponding to the new version. This guarantees that there is no instant during an update where the object may disappear from the data store. For example, in Figure 3.3, the update modifies the object to move it from the blue server to the green server. The value-dependent chain for the update passes through both

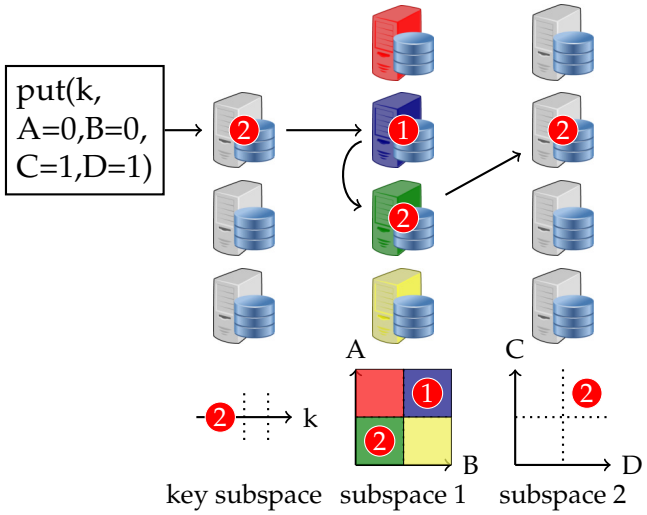


Figure 3.3: HyperDex's replication protocol propagates along value-dependent chains. Each update has a value-dependent chain that is determined solely by objects' current and previous values and the hyperspace mapping.

servers and is determined by the existing value and the new value to be written. The update will result in the object being stored on the green server, and being subsequently removed from the blue server as acknowledgments propagate in reverse.

Successive updates to an object will construct chains which overlap in each subspace. Consequently, concurrent updates may arrive out of order at each of these points of overlap. If handled improperly, such races could lead to inconsistent, out-of-order updates. Value-dependent chains efficiently handle this case by dictating that the point leader embed, in each update, dependency information which specifies the order in which updates are applied. Specifically, the point leader embeds a version number for the update, and the version number, hash and type of the previous update. This results in HyperDex performing a read before each write to construct the dynamic chain.

By design, HyperDex supports destructive operations that remove all state

pertaining to deleted objects. Examples of destructive operations include deleting objects and the cleanup associated with object relocation. Such operations must be carefully managed to ensure that subsequent operations get applied correctly. For instance, consider a `delete` followed by a `put`. The `del` operation should remove all state, while the `put` places state in each subspace. Absent synchronization, concurrent `del` and `put` operations could race in the network, and there would be no guarantee that operations are applied in-order: A late-arriving message for the `put` could improperly restore the deleted object, while a late-arriving message for the `del` could improperly delete a newly written object. Value-dependent chains ensure that concurrently issued destructive operations are correctly ordered on all servers. Each server independently delays operations which depend upon a destructive operation until the destructive operation, and all that came before it, are acknowledged. This ensures that at most one destructive operation may be in-flight at any one time and guarantees that they will be ordered correctly. The delay for each message is bounded by the length of chains, and the number of concurrent operations.

3.4.2 Fault Tolerance

To guard against server failures, HyperDex inlines a chain of replicas within each region. The replicas acquire and maintain their state by being incorporated into value-dependent chains. In particular, each region has $f + 1$ replicas which appear as a block in the value-dependent chain. For example, we can extend the layout of Figure 3.3 to tolerate one failure by introducing additional hosts. As with regular chain replication [121], new replicas are introduced at the tail of the region's chain, and servers are bumped forward in the chain as other servers fail. And as with regular chain replication, this transition will be performed

without compromising strong consistency.

3.4.3 Consistency Guarantees

Overall, the preceding protocol ensures that HyperDex provides strong guarantees for applications. The specific guarantees made by HyperDex are:

Key Consistency All actions which operate on a specific key (e.g., `get` and `put`) are linearizable [56] with all operations on all keys. This guarantees that all clients of HyperDex will observe updates in the same order.

Search Consistency HyperDex ensures that a search will return all objects that were committed at the time of search and not updated during the search. An application whose `put` succeeds is guaranteed to see the object in a future search if no other applications write the object. In the presence of concurrent updates, a search may return both the committed version, and the newly updated version of an object matching the search, either version, or neither.

HyperDex provides the strongest form of consistency for key operations, and a conservative and predictable consistency guarantees for search operations.

3.5 Evaluation

HyperDex was implemented to support all features described in this chapter. At the time of its initial publication [41], HyperDex consisted of approximately 44,000 lines of code. Since its original open-source release the code has gone through extensive changes and currently consists of approximately 100,000 lines of code. This section evaluates an early release of HyperDex and comparable systems released in the same time frame.

This section examines the performance of HyperDex on both a small and medium-size computational clusters and reports the performance of each deployment using the Yahoo! Cloud Serving Benchmark (YCSB) [30], an industry-standard benchmark for cloud storage performance. The evaluation also examines the performance of HyperDex's basic operations, specifically, `get`, `put`, and `search`, using targeted micro-benchmarks. These micro-benchmarks isolate specific components and help expose the performance impact of design decisions. For both YCSB and the micro-benchmarks, HyperDex is compared to Cassandra [65], a popular key-value store write-heavy workloads and MongoDB [82], a distributed document database.

The performance benchmarks are executed on a small, dedicated lab-size cluster in order to avoid confounding issues arising from sharing a virtualized platform, while the scalability benchmarks are executed on the VICCI [92] testbed. The dedicated cluster consists of fourteen nodes, each of which is equipped with two Intel Xeon $2.5\ \mathrm{GHz}$ E5420 processors, $16\ \mathrm{GB}$ of RAM, and a $500\ \mathrm{GB}$ SATA $3.0\ \mathrm{Gbit/s}$ hard disk operating at 7200 RPM. All nodes are running 64-bit Debian 6 with the Linux 2.6.32 kernel. A single gigabit Ethernet switch connects all fourteen machines. Each machine was configured to run Cassandra version 0.7.3, MongoDB version 2.0.0, and HyperDex.

For all tests, the storage systems are configured to provide sufficient replication to tolerate one node failure. Each system was configured to use its default consistency settings. Specifically, both Cassandra and MongoDB provide weak consistency and fault-tolerance guarantees; because acknowledgments are generated without full replication, small numbers of failures can lead to data loss. In contrast, HyperDex utilizes value-depending chaining and, as a result, always provides clients with strong consistency and fault-tolerance, even in the

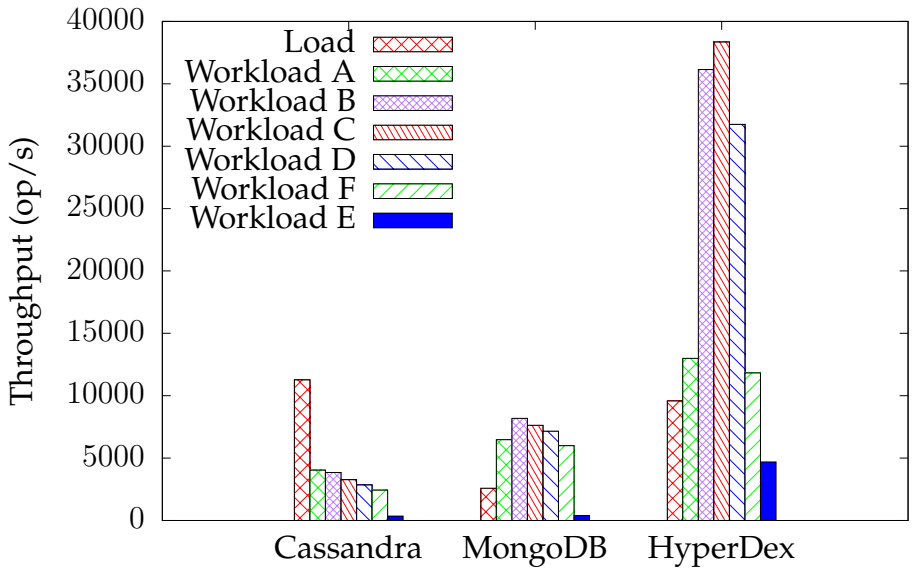


Figure 3.4: Average throughput for a variety of real-world workloads specified by the Yahoo! Cloud Serving Benchmark. HyperDex is 3-13 times faster than Cassandra and 2-12 times faster than MongoDB. Workload E is a search-heavy workload, where HyperDex outperforms other systems by more than an order of magnitude.

presence of failures. Since MongoDB allocates replicas in pairs, all clusters were provisioned with twelve machines for the storage nodes, one machine for the clients, and, where applicable, one node for the coordinator. HyperDex is configured with two subspaces in addition to the key subspace to accommodate all ten attributes in the YCSB dataset.

3.5.1 Get/Put Performance

High `get/put` performance is paramount to any cloud-based storage system. YCSB provides six different workloads that exercise the storage system with a mixture of request types and object distributions resembling real-world applications (Table 3.1). In all YCSB experiments, the database is preloaded with 10,000,000 objects and each operation selects the object and operation type in

Name	Workload	Key Distribution	Sample Application
A	50% Read/50% Update	Zipf	Session Store
B	95% Read/5% Update	Zipf	Photo Tagging
C	100% Read	Zipf	User Profile Cache
D	95% Read/5% Insert	Temporal	User Status Updates
E	95% Scan/5% Insert	Zipf	Threaded Conversations
F	50% Read/50% Read-Modify-Write	Zipf	User Database

Table 3.1: The six workloads specified by the Yahoo! Cloud Serving Benchmark. These workloads model several applications found at Yahoo! Each workload was tested using the same YCSB driver program and system-specific Java bindings. Each object has ten attributes which total $1\ \mathrm{kB}$ in size.

accordance with the workload specification. Figure 3.4 shows the throughput achieved by each system across the YCSB workloads. HyperDex provides throughput that is between a factor of two to thirteen higher than the other systems. The largest performance gains come from improvements in `search` performance. Significant improvements in `get/put` performance is attributable mostly to the efficient handling of `get` operations in HyperDex. Our implementation demonstrates that the hyperspace construction and maintenance can be realized efficiently.

HyperDex's performance is predictable: all reads complete in under $1\ \mathrm{ms}$, while a majority of writes complete in under $3\ \mathrm{ms}$. Cassandra's latency distributions follow a similar trend for workloads B, C, D and F and show a slightly different trend for workload A. MongoDB, on the other hand, exhibits lower latency than Cassandra for all operations. For all workloads in YCSB, Hyper-

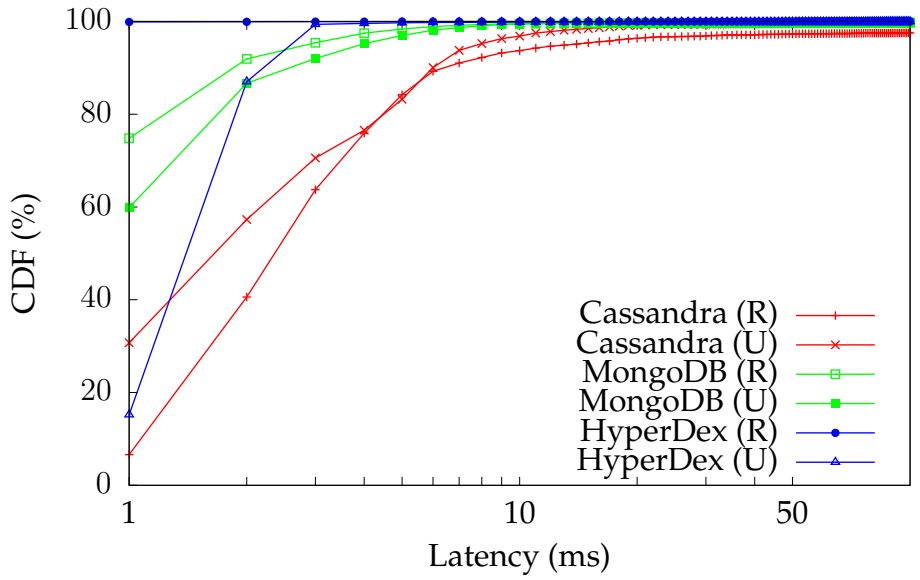


Figure 3.5: `GET/PUT` performance. Latency distribution for Workload A (50% reads, 50% updates, Zipf distribution).

Dex completes 99% of operations sooner than either Cassandra and MongoDB. Figures 3.5 and 3.6 show the latency distributions for workloads A and B respectively.

For completeness, this subsection presents the performance of all three systems on a write-heavy workload. Figure 3.7 shows the latency distribution for inserting 10,000,000 objects to set up the YCSB tests. Consistency guarantees have a significant effect on the `put` latency. MongoDB's default behavior considers a `put` complete when the request has been queued in a client's send buffer, even before it has been seen by any server. Cassandra's default behavior considers a `put` complete when it is queued in the filesystem cache of just one replica. Unlike these systems, the latency of a HyperDex `put` operation includes the time taken to fully replicate the object on $f + 1$ servers. Because MongoDB does not wait for full fault tolerance, it is able to complete a majority of operations in less than 1 ms; however, it exhibits a long-tail (Figure 3.7) that adversely

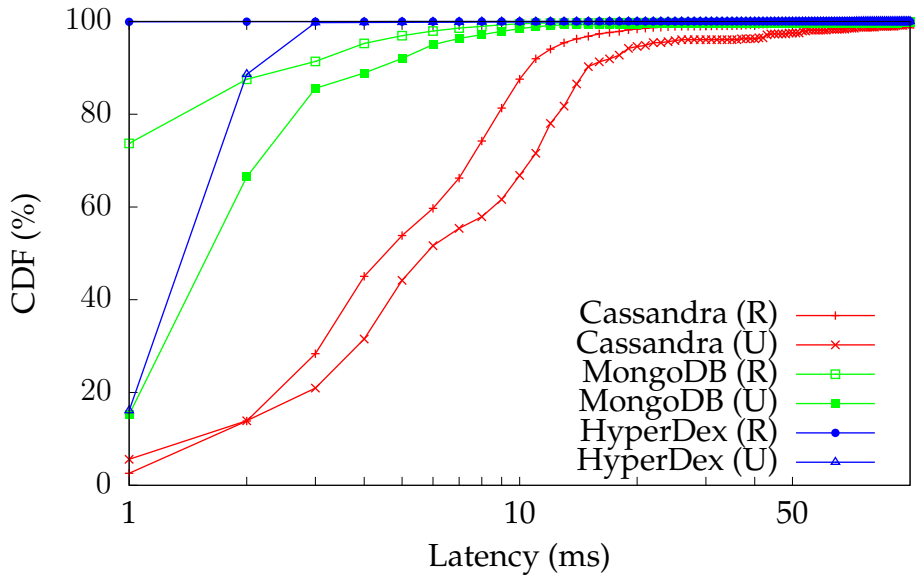


Figure 3.6: `GET/PUT` performance. Latency distribution for Workload B (95% reads, 5% updates, Zipf distribution). HyperDex maintains low latency for reads and writes.

impacts average throughput. Similarly, Cassandra completes most operations in less than 2 ms. Despite its stronger fault-tolerance guarantees, HyperDex completes 99% of its operations in less than 2 ms.

3.5.2 Search vs. Scan

Unlike existing key-value stores, HyperDex is architected from the ground-up to perform `search` operations efficiently. Current applications that rely on existing key-value stores emulate search functionality by embedding additional information about other attributes in the key itself. For example, applications typically group logically related objects by using a shared prefix in the key of each object, and then rely upon the key-value store to locate keys with the common prefix. In Cassandra, this operation is efficient because keys are stored in sorted order, and returning all logically grouped keys is an efficient linear

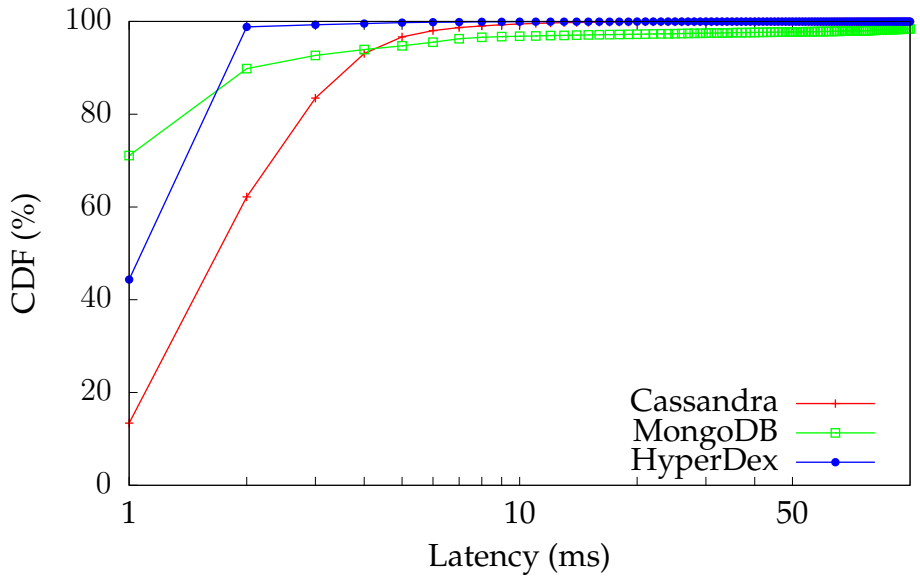


Figure 3.7: `PUT` performance. Latency distribution for 10,000,000 operations consisting of 100% insertions. Each operation created a new object under a unique, previously unused key. Although fast, HyperDex's minimum latency is bounded by the length of the value-dependent chains.

scan. Fittingly, the YCSB benchmark calls this a `scan` operation. HyperDex's `search` functionality is a strict superset of the `scan` operation. Rather than using a shared prefix to support scans, HyperDex stores, for each object, the prefix and suffix of the key as two secondary attributes. Scans are then implemented as a multi-attribute search that exactly matches a provided prefix value and a provided range of suffix values. Thus, all YCSB benchmarks involving a `scan` operation operate on *secondary attributes* in HyperDex, but operate on the *key* for other systems.

Despite operating on secondary attributes instead of the key, HyperDex outperforms the other systems by an order of magnitude for `scan` operations (Figure 3.8). Seventy five percent of search operations complete in less than 2 ms, and nearly all complete in less than 6 ms. Cassandra sorts data according to the primary key and is therefore able to retrieve matching items relatively quickly.

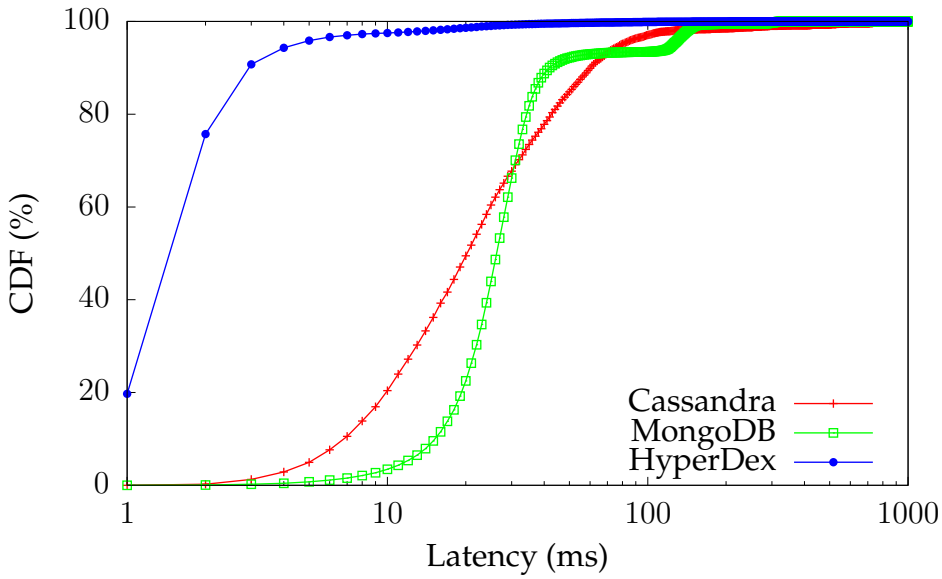


Figure 3.8: `SEARCH` performance. Latency distribution for 10,000 operations consisting of 95% range queries and 5% inserts with keys selected from a Zipf distribution. HyperDex is able to offer significantly lower latency for non-primary key range queries than the other systems are able to offer for primary-key range queries.

Although one could alter YCSB to use Cassandra's secondary indexing schemes instead of key-based operations, the result would be strictly worse than what is reported for primary key operations. MongoDB's sharding maintains an index; consequently, `scan` operations in MongoDB are relatively fast. The `search` performance of HyperDex is not attributable to an efficient implementation as `search` is more than an order of magnitude faster in HyperDex, which eclipses the 2-4× performance advantage observed for `get/put` throughput. Hyperspace hashing in HyperDex ensures that search results are located on a small number of servers; this enables effective pruning of the search space and allows each search to complete by contacting exactly one host in these experiments.

An additional benefit of HyperDex's aggressive search pruning is the relatively low latency overhead associated with `search` operations. Figure 3.9

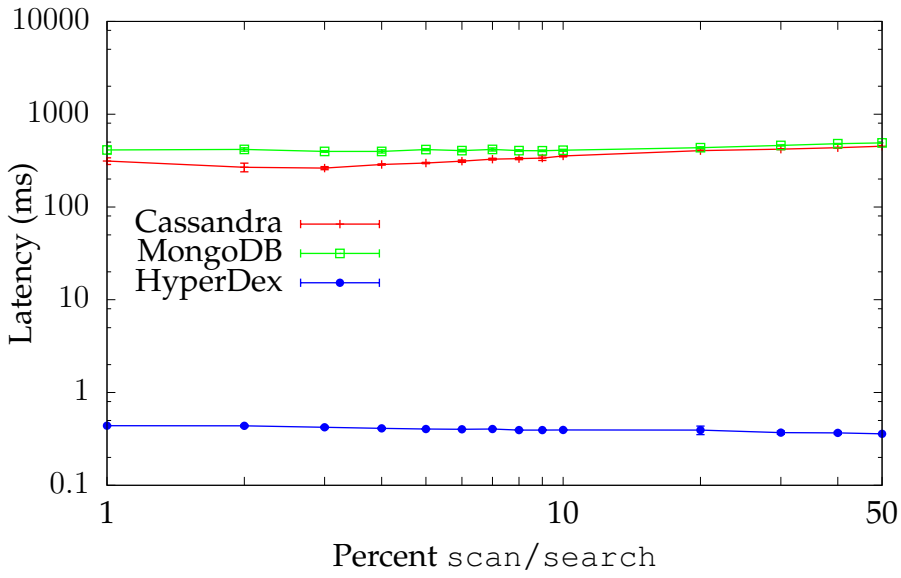


Figure 3.9: The effect of an increasing scan workload on latency. HyperDex performs significantly better than the other systems even as the scan workload begins to approach 50%.

shows the average latency of a single `scan` operation as the total number of `scan` operations performed increases. In this test, searches were constructed by choosing the lower bound of the range uniformly at random from the set of possible values, as opposed to workload E which uses a Zipf distribution to select objects. Using a uniform distribution ensures random access, and mitigates the effects of object caching. HyperDex consistently offers low latency for `search`-heavy workloads.

A critical parameter that affects HyperDex's search performance is the number of subspaces in a HyperDex table. Increasing the number of subspaces leads to additional opportunities for pruning the search space for *search* operations, but simultaneously requires longer value-dependent chains that result in higher `put` latencies. In Figure 3.10, we explore the tradeoff using between zero and ten additional subspaces beyond the mandatory key subspace. Note that adding

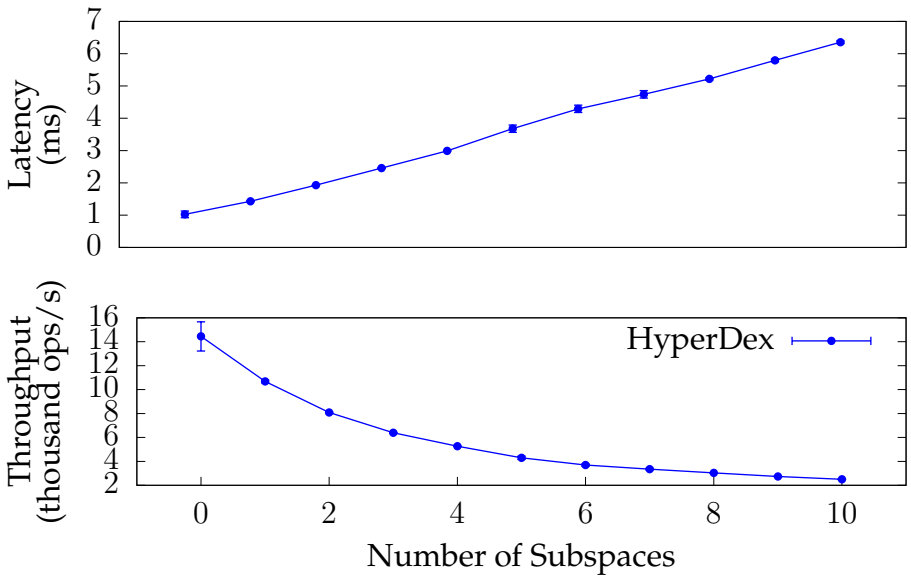


Figure 3.10: Latency and throughput for `put` operations as a function of non-key subspaces The error bars indicate standard deviation from 10 experiments. Latency increases linearly in the length of the chain, while throughput decreases proportionally. In sample applications built with HyperDex, all tables have three or fewer subspaces.

ten additional subspaces increases the value-dependent chain to be at least 22 nodes long. As expected, HyperDex's `put` latency increases linearly with each additional subspace.

3.5.3 Scalability

We have deployed HyperDex on the VICCI [92] test-bed to evaluate its performance in an environment representative of the cloud. Each VICCI cluster has 70 Dell R410 PowerEdge servers, each of which has 2 Intel Xeon X5650 CPUs, 48 GB of RAM, three 1 TB hard drives, and two 1 Gbit ethernet ports. Users are provided with an isolated virtual machine for conducting experiments. Each virtual machine comes preinstalled with Fedora 12 and runs the 2.6.32 Linux kernel.

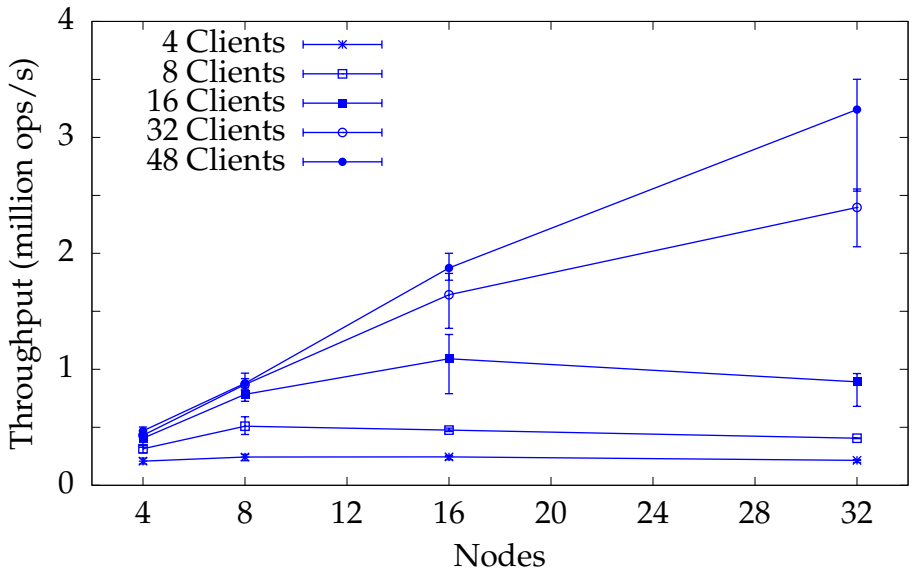


Figure 3.11: HyperDex scales horizontally. As more servers are added, aggregate throughput increases linearly. Each point represents the average throughput of the system in steady state over 30 second windows. The error bars show the 5^{th} and 95^{th} percentiles.

Figure 3.11 shows the performance of a HyperDex cluster as the cluster increases in size. Increasing the number of servers in the cluster provides HyperDex with additional resources and leads to a proportional increase in throughput. The experiment explores the change in system throughput as resources are added to the cluster for a static amount of offered load. As expected, HyperDex scales linearly as resources are added to the cluster. Each point in the graph represents the average throughput observed over a 30 second window and the error bars show the 5^{th} and 95^{th} percentiles observed over any 1-second window. At its peak, HyperDex is able to average 3.2 million operations per second.

The workload for Figure 3.11 is a 95% read, 5% write workload operating on 8 B keys and 64 B values. The measurements reported are taken in steady state, with clients randomly generating requests according to the workload. This workload and measurement style reflects the workload likely to be encountered

in a web application. The reported measurements exclude the warm-up time for the system. In all experiments, 15 seconds was sufficient to achieve steady state. Clients operate in parallel, and are run on separate machines from the servers in all but the largest configurations. Clients issue requests in parallel, and each client maintains an average of 1,000 outstanding requests per server. Increasing the number of clients does not significantly impact the achievable average throughput.

This experiment shows that a medium-sized HyperDex cluster is able to achieve high throughput for realistically sized deployments. Additional resources allow the cluster to provide proportionally better throughput.

3.6 Conclusions

This chapter described HyperDex, a NoSQL storage system that combines strong consistency guarantees with high availability in the presence of failures and partitions affecting up to a threshold of servers. In addition, HyperDex provides an efficient search primitive for retrieving objects through their secondary attributes. It achieves this extended functionality through *hyperspace hashing*, in which multi-attribute objects are deterministically mapped to coordinates in a low dimension Euclidean space. This mapping leads to efficient implementations for key-based retrieval, partially-specified searches and range-queries. HyperDex's replication protocol enables the system to provide strong consistency without sacrificing performance. Industry-standard benchmarks show that the system is practical and efficient.

CHAPTER 4

INTRA-DATA CENTER TRANSACTIONS IN WARP

4.1 Introduction

NoSQL systems have become the de facto back-end for modern applications because they allow unprecedented performance at large scale. The defining characteristic of these systems is their distributed architecture, where the system shards data across multiple servers to improve scalability. To further improve scalability, these systems typically avoid cross-server communication, which makes it difficult to implement ACID transactions.

Yet, distributed transactions require coordination among multiple servers. In traditional RDBMSs, transaction managers coordinate clients with servers, and ensure that all participants in multi-phase commit protocols run in lock-step. Such transaction managers constitute bottlenecks, and modern NoSQL systems have eschewed them for more concurrent implementations. Scatter [50] and Google's Megastore [14] shard the data across different Paxos groups based on their key, thereby gaining scalability, but incur higher coordination costs for actions that span multiple groups, and require that operations on the same group be sequenced by the same Paxos instance. Google's Spanner [31] combines traditional locking techniques with a novel TrueTime API to provide fast read-write transactions, and lock-free reads. Many recent systems propose moving pieces of the transactional programs themselves to the server. Calvin [120] serializes all transaction inputs via a consensus protocol, and then deterministically executes application code on servers. Lynx [126] and Rococo [84] break down the transaction into multiple atomic fragments, and employ static analysis to detect opportunities for reordering the shipped code components. Both

techniques rely upon a priori knowledge and analysis of the transactions.

This chapter introduces Warp, a NoSQL system that enables efficient multi-key transactions with ACID semantics. Warp offers, to our knowledge, the highest degree of concurrency in a general purpose serializable NoSQL data store. Further, it achieves throughput far in excess of previous systems, approaching 75% of the throughput of the system on which it builds. The key insight that enables these performance improvements is a commit protocol called *linear transactions*, which allows the system to order transactions on-the-fly without any prior static analysis, and without moving code to the server.

Three techniques, working in concert, enable linear transactions to achieve its high scalability and performance. First, linear transactions reduce coordination costs by reducing the number of transactions that are ordered to the minimum necessary to enforce serializability. Transactions that operate on disjoint data or whose executions do not overlap in time will incur zero coordination costs. Warp orders only those transactions that concurrently operate on overlapping data, and does so with minimal overhead.

Second, linear transactions achieve scalability by arranging servers into data-dependent, dynamically-determined chains, where each chain contains, solely, those servers which store data affected by the transaction. This structure, avoids bottlenecks at transaction managers and improves performance by cutting communication costs.

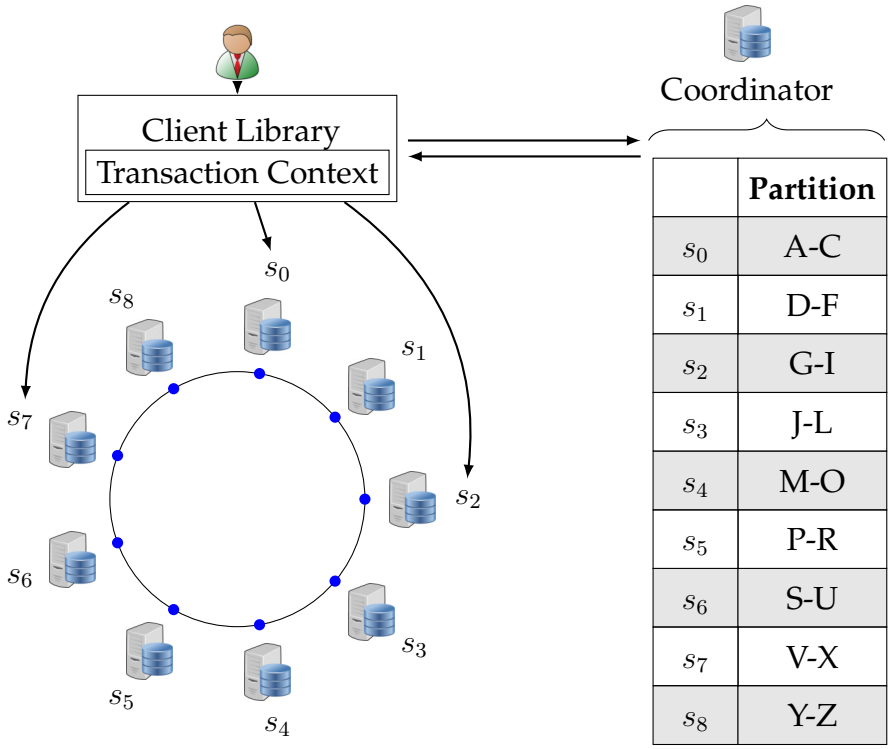
Finally, linear transactions improve performance by allowing multiple overlapping transactions to proceed in parallel under the right conditions. Locking protocols inherently limits concurrency, while traditional optimistic two-phase protocols can only prepare one transaction per key at a time, because all reads must be validated in the first phase before the second phase may begin. In con-

trast, Warp enables multiple transactions to prepare simultaneously by taking advantage of the inherent serialization in its data-dependent chains.

Overall, this chapter makes three contributions. First, we outline a novel protocol for providing efficient, one-copy serializable transactions on a distributed, sharded data store. Our protocol provides an unprecedented level of concurrency and scalability without any synchronicity assumptions or static analysis. Second, we describe our implementation of the commercially available Warp key-value store, including the design of the client. The system has been fully implemented and provides language bindings for C, C++, Python, Java, Ruby, Go, and Node.JS. Third, we show through macro- and micro-benchmarks that Warp can provide higher throughput than alternative designs, with fewer aborted transactions. Specifically, Warp achieves a throughput that is $4\times$ higher, with $5\times$ lower latency, than mini-transactions [4], the closest existing approach, on the TPC-C benchmark. The system achieves 75% the throughput of the underlying non-transactional key-value store upon which Warp builds.

4.2 Design

Warp builds upon the HyperDex [41] key-value store by modifying the existing components to provide transactional guarantees. Warp's architecture is based on HyperDex. To that end, Warp consists of three components: the coordinator, clients, and storage servers. The coordinator maintains the meta-state for the system, specifically, the partitioning of the key space across storage servers. Clients issue requests directly to the storage servers, where a request may affect a single object, or span multiple objects. Each storage server maintains a subset of keys in the system; collectively, the storage servers hold all data stored by the system. Figure 4.1 illustrates Warp's overall architecture.



	Partition
s_0	A-C
s_1	D-F
s_2	G-I
s_3	J-L
s_4	M-O
s_5	P-R
s_6	S-U
s_7	V-X
s_8	Y-Z

Figure 4.1: Warp's architecture consists of storage servers, the coordinator, and the client library. The coordinator maintains the partitioning of the key space across servers, and supplies this mapping to the client library. The client library uses this mapping to directly contact storage servers.

The Warp client library provides isolation by optimistically performing read and write operations against local state, and verifying that this state remains the same at commit time. To perform a read, the library retrieves the requested data from the storage servers and caches the object within the local transaction's state, called the *transaction context*. Subsequent reads within the transaction will be satisfied by the transaction context, if possible. Write operations executed within the transaction are not visible on the servers immediately. Instead, they are saved to the transaction context to be written at commit time. Multiple writes to the same key will overwrite the locally saved object. Storage servers are unaware of any modifications written within a transaction until the

client commits the transaction. To commit the transaction, the library submits the set of all objects read and all objects written to the storage servers using the linear transactions commit protocol.

4.2.1 Commit Protocol

The linear transactions commit protocol processes clients' transactions, and ensures that they either commit in an atomic, serializable fashion, or abort with no effect. The protocol does this for each transaction by simultaneously validating the values optimistically read by the client library, and ordering the transaction with respect to other concurrently executing transactions. While it is relatively easy to validate a transaction by comparing the values observed by the client to the latest values in the data store, it is considerably more difficult to order transactions across multiple servers. The former is a local check that each server may independently perform during transaction processing, while the latter requires that multiple servers coordinate to agree upon the serializable order of transactions.

The key insight of the linear transactions protocol is to arrange the servers for a transaction into a chain, and to validate and order transactions using a dynamically-determined number of passes through this chain. Compared to traditional commit protocols which use fan-out/fan-in communication patterns, linear transactions pass messages forward or backward between adjacent nodes in the chain. This ensures that there is at most one server actively processing each transaction at any one time. By limiting the parallelism present in a single transaction, linear transactions enable each server to locally make a binding decision about the fate of the transaction they are processing, and propagate that decision to the next server in the chain. Globally, this enables multiple trans-

actions which modify the same data to execute in parallel—transactions whose execution other techniques would serialize—because each pair of concurrent transactions is ordered by exactly one server that can decide their order without communicating with other servers. Any decision made by a server will be carried to, and enforced by, the remaining servers in the chain.

The protocol consists of multiple distinct processes that work in concert to commit transactions. To commit the transaction, the client library determines the chain of servers which process the transaction's commit. These are *only* those servers that house the data involved in the transaction. The servers in this chain follow simple rules to commit the transactions: a server passes a transaction forward in the chain only when the server may commit that transaction; otherwise, the server sends either an abort or a retry request backward in the chain. Transactions will be aborted when they fail to validate the clients' reads, and will be retried to ensure the order between concurrent transactions is serializable. When the transaction passes completely through the chain in the forward direction, the last server in the chain finalizes the transaction by sending a commit message backwards through the chain. This commit message instructs servers to persist the transactions to disk, and to clean up any transient state related to the transaction.

Chain Construction

Clients use the transaction's context to construct a chain to commit the transaction. To ensure that servers process transactions' operations in a predictable order, the client library sorts the keys read or written by a transaction in lexicographical order, and maps this sorted list onto a set of storage servers. Because sorting is a deterministic process, transactions with multiple keys in common

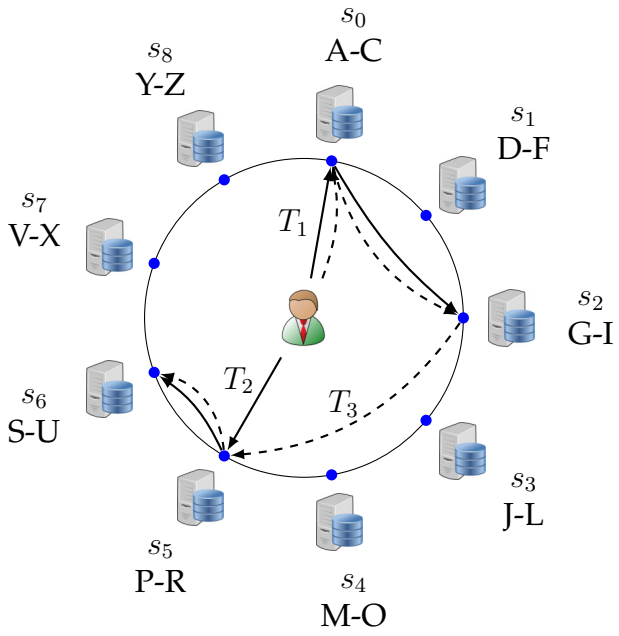


Figure 4.2: Clients deterministically construct dynamic chains based upon the keys read and written by transactions. In this example, a client submits T_1, T_2, and T_3. Transactions T_1 and T_2 operate on disjoint keys, $\{k_A, k_H\}$ and $\{k_P, k_T\}$ respectively. T_3 touches all four keys and forms a chain that includes the chains of T_1 and T_2
.

pass through their shared set of servers in the same order.

Figure 4.2 shows how transactions that read and write the same keys have overlapping chains. Transaction T_1 reads key k_H and writes key k_A, while transaction T_2 reads keys k_P and k_T. Transaction T_3 writes keys k_A, k_H, k_P, and k_T. The object-to-server mapping dictates that T_1's chain pass through servers s_0 and s_2 because these servers hold objects k_A and k_H respectively. Similarly, T_2 forms a chain through s_5 and s_6. T_3 writes the same keys touched by T_1 and T_2 and has a chain that passes through the same servers as transactions T_1, and T_2.

Constructing chains in this way makes it straightforward to order transactions that concurrently operate on the same data. The first server in common between two transactions' chains can order any two overlapping transactions, and

notify all subsequent servers in both chains. The mapping maintained by the coordinator ensures that transactions with data in common will pass through a common set of same storage servers, even when the mapping is updated to reflect system membership changes. Inversely, when two chains do not overlap, there is no need to directly order their transactions, because they necessarily operate on disjoint data.

Validation

The validation step ensures that values previously read by the client remain unchanged until the transaction commits. To do this, servers ensure that the value the client read during its transaction is the same value currently in the data store, and that no concurrent transactions change the value that the client read. Specifically, servers check each transaction to ensure that it does not read values written by, or write values read by, previously validated transactions. Servers also check each value against the latest value in their local store to ensure that the value was not changed by a previously-committed transaction. Thus, linear transactions employ optimistic concurrency control [64, 118].

Servers perform validation for each transaction before forwarding it to subsequent servers in the chain. This ensures that at any step in a transactions' processing, a prefix of the chain guarantees that the transaction is valid and will remain valid until the transaction commits or aborts. Storage servers will abort subsequent transactions whose writes would invalidate previously valid transactions. Consequently, when a transaction reaches the last server in the chain, that server can decide to commit or abort the transaction without consulting any other server—the protocol guarantees to the server that every previous server will be able to commit the transaction.

When a server determines that a transaction does not validate, the server aborts the transaction by sending an abort message backwards through the chain. Each server in the prefix aborts the transaction and forwards the abort message until the message reaches the client. These servers remove the transaction from their local state, enabling other transactions to validate in its place.

Ordering

The central task of the linear transactions protocol is to establish a serializable order across all validated transactions. While the protocol could simply serialize all transactions—which would maximize spurious coordination—it instead relies upon the observation that a set of transactions are serializable if the dependency graph of their relative orders is free of cycles. Each edge in this graph specifies a pair of transactions and the relative order between them. We refer to the transactions at each endpoint of an edge as a *conflicting pair*, because one transaction contains a write operation on a key which was read or written by the other. Consequently, every conflicting pair has at least one, and possibly several, keys that are common to both transactions.

The difficulty in ordering these conflicting pairs lies not in resolving pairwise relationships, but in ensuring that every pairwise ordering is consistent with the globally serializable order. Resolving the order across multiple pairs is a considerably more complex task, because interactions between transactions can span multiple servers. In these cases, it is possible that no single server would have have the requisite view to detect and prevent a cycle in the graph. For example, imagine three transactions across keys k_A, k_B, and k_C, where each transaction writes to a different pair of the keys: (k_A, k_B), (k_B, k_C), and (k_C, k_A). If the system only ordered the transactions pairwise, the three transactions could be

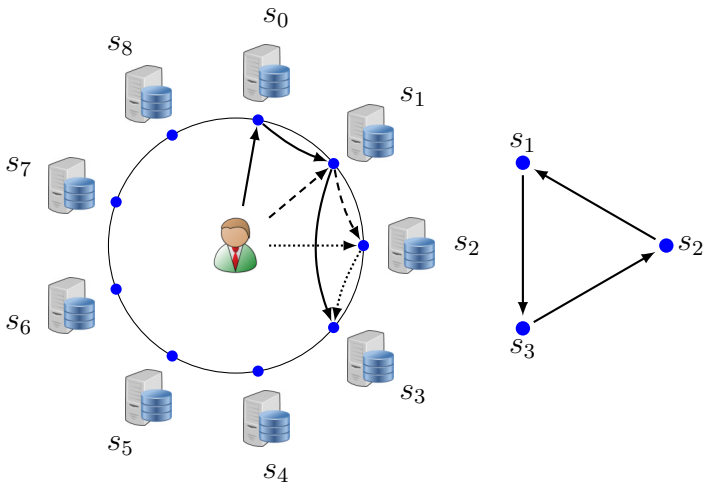


Figure 4.3: Ordering transactions in a serializable fashion across multiple servers is difficult because of the possibility of distributed cycles. In this figure, the three transaction's chains overlap on servers s_1, s_2, s_3, but that no one server handles all three transactions. The transaction chaining protocol prevents these transactions from forming the cycle shown on the right.

committed in a non-serializable order, because no single key is written by all three transactions. Figure 4.3 depicts this example and highlights the problematic execution that results in a cycle between the transactions.

Servers ensure that all transactions commit in a serializable order across servers by embedding ordering information, called *mediator tokens*, into transactions. Mediator tokens are integer values that are assigned to transactions by the heads of chains. Because mediator tokens are integers, servers may determine the relative order of conflicting pair by comparing their mediator tokens. A simple invariant that ensures serializability is to commit conflicting pairs in the order specified by their mediator tokens. For example, if the mediator tokens for the conflicting pair (T_X, T_Y) have the relationship $\text{mediator}(T_X) < \text{mediator}(T_Y)$, then all servers order the transactions such that T_X commits before T_Y.

The linear transactions protocol relies on this invariant to order transactions. Upon receipt of a transaction passing forward through the chain, a server com-

pares the transaction's mediator token to the largest mediator token across all transactions that previously read or wrote any of the current transaction's objects. If the current mediator token is larger than the previous token, the transaction is forwarded to the next server in the chain. If, however, the mediator token is less than the previous token, a "retry" message is sent backwards in the chain to the head, where the transaction will be retried with a larger mediator token.

Generating Mediator Tokens

At first glance, mediator tokens may resemble timestamps that are typically used by transaction commit protocols, but mediator tokens are significantly more flexible. Systems based upon timestamps, whether they be logical timestamps [68] or synchronized wall clocks, impose restrictions on how timestamps may be generated. Namely, timestamps must be generated in a monotonic fashion so that the system never moves backward in time. Failing to preserve the monotonicity of timestamps would permit transactions to commit in an unserializable fashion. Mediator tokens impose no such restrictions on token generation.

The flexibility of mediator tokens permits a wide array of implementation strategies. For example, a simple token generation strategy would be to always set a transaction's initial token to zero, choosing successively larger values on each subsequent pass. Another strategy would be to generate tokens at random and ensure that each subsequent pass draws from a range of tokens that are strictly greater than the token of the previous pass. A strategy that limits the number of retries is for each server to maintain a counter to generate mediator tokens. Servers may generate a new mediator token by reading the counter's

current value and incrementing the counter. When a server sees a mediator token that is greater than the next value to be generated by its counter, it may advance the counter to be greater than this other token. Because mediator tokens are flexible, storage servers do not need to carefully manage or preserve this counter during server failover, and do not need to synchronize counters across servers.

4.2.2 Fault Tolerance and Durability

In a large-scale deployment, failures are inevitable. Linear transactions accommodate a natural way to overcome such failures. Specifically, linear transactions permit a subchain of $f + 1$ replicas to be inlined into the longer chain in place of a single data server. This allows the system to remain available despite up to f failures within a subchain. Chain replication maintains a well-ordered series of updates within each subchain. Operations that traverse the linear transaction chain in the forward direction pass forward through all inlined chains. Likewise, operations that traverse the chain in reverse traverse inlined chains in reverse.

The notion of fault-tolerance provided by linear transactions is different from the notion of durability within traditional databases. While durability ensures that data may be re-read from disk after a failure, the system remains unavailable during the failure and recovery period; in contrast, linear transactions' fault tolerance mechanism ensures that the system remains available so long as the number of failures remains below the configured threshold.

4.2.3 Atomicity, Consistency, Isolation

The protocol guarantees atomicity, consistency, and isolation for all transactions. These properties naturally follow from the one-copy serializability upheld by the protocol. Each transaction completes in its entirety at a well-defined point in the partial order, where its effects are either completely visible to subsequent transactions, or it aborts without effect. Every server ensures that the stored objects are well-formed and match their data types. Overall, linear transactions guarantee that operations within a transaction execute with mutual exclusion from each other, as if there were a single giant lock protecting the database.

4.2.4 Correctness

By leveraging a fault tolerant system coordinator, linear transactions uphold both liveness and safety in the presence of up to f faults. Specifically, linear transactions maintain serializability at all times, and will eventually commit or abort every transaction assuming at most f of the $f + 1$ replicas of the data remain non-faulty. In this section we demonstrate how the linear transactions protocol maintains these safety and liveness properties.

Safety: The linear transactions protocol provides serializability by ensuring that the final system state is equivalent to a serial schedule. The protocol upholds this guarantee by ensuring that the dependency graph across all transactions is acyclic. Intuitively, every conflicting pair directly corresponds to an edge in this graph, while the mediator tokens enforce the anti-cycle property.

Because non-conflicting pairs operate on disjoint sets of data, the conflicting pairs are the only transaction pairs whose order must be carefully managed. A conflicting pair of transactions is committed in the same order on all common

servers in order to ensure that operations to their shared state are applied in the same order. Servers use mediator tokens to decide the commit order for transactions in a conflicting pair; every server will enforce the order specified by the transactions' tokens.

Globally, mediator tokens preserve transitive relationships across transactions, ensuring that cycles cannot arise in the dependency graph. The dependency graph is structured such that each edge is a conflicting pair (T_x, T_y) such that either $\text{mediator}(T_x) < \text{mediator}(T_y)$ or $\text{mediator}(T_x) > \text{mediator}(T_y)$. In the former case, the graph will contain an edge $T_x \rightsquigarrow T_y$, while in the latter case, the graph will contain edge $T_y \rightsquigarrow T_x$. Any directed path will consist of edges such that the mediator token for the source is less than the mediator token for the destination. Transitively, the start of the path must have a mediator token less than the end of the path. Thus, it is impossible for the graph to contain a cycle, because a cycle would imply that there exists a directed path—and thus, a directed edge—from a transaction with a higher mediator token to a transaction with a lower mediator token. Because the system prohibits committing transactions in an order that contradicts the ordering established by their mediator tokens, it is impossible for such an edge, and thus a cycle, to exist.

Liveness: The protocol remains available for processing transactions during a bounded number of server failures. Specifically, the protocol will always be able to commit or abort a transaction so long as at most f servers fail of the $f+1$ servers assigned to replicate each key. To enable the system to detect and correct for these failures, linear transactions make use of a fault tolerant coordinator, which may be built using standard techniques [5, 22, 58]. This coordinator acts as a shepherd for the system, guiding it toward a stable state, even as servers fail or become otherwise unavailable.

The system overcomes failures by removing failed servers from the chains for actively propagating transactions. For each failure, the coordinator issues a new configuration that lists the server as failed. Non-faulty servers may consult this new configuration to determine which currently outstanding transactions contain the faulty server as the next hop in the forward direction. These transactions are retransmitted to move the transaction toward a commit or abort state. To prevent duplicate messages from affecting correctness, servers maintain a list of committed and aborted transactions. Upon receipt of a retransmitted forward-bound message, a server will first consult this list and answer with a commit or abort message if appropriate. Otherwise, the server processes the message to move the transaction closer to committing; typically this will entail sending another message forward in the chain, or waiting for a previously sent message to return "commit" or "abort". Overall, the coordinator and servers will repeat this process as necessary until transactions eventually commit or abort.

4.3 Implementation

We have fully implemented the system described in this chapter. The code base consists of 130,000 lines of code, approximately 15,000 lines of which are devoted to processing transactions. The Warp distribution provides bindings for C, C++, Python, Ruby, Java, Go, and Node.JS and supports a rich API that goes well beyond the simple `get`/`put` interface of typical key-value stores. A system of virtual servers maps a small number of servers to a larger number of partitions, permitting the system to reassign partitions to servers without repartitioning the data. The implementation uses a replicated state machine as the coordinator to ensure that there are no single points of failure.

4.3.1 Rich API

Warp naturally supports an expanded API that enables complex applications. The expanded API includes support for rich data structures, multiple independent schemas, and nested transactions.

Data Structures

The discussion in Section 4.2 presented all operations in a linear transaction as either a read or a write on an arbitrary string, but our implementation goes much further to support many data structures commonly used in modern applications. Warp provides programmers with integer, float, list, set, map, and document types as well as atomic operations on each of these types that enable fast concurrent operation. For example, it is possible to atomically add an element to a list, or perform arithmetic on an integer type. A write, then, may consist of any of these atomic operations and is not limited to simply overwriting the previous value. These atomic operations are especially useful for cases where linear transactions enable low abort and retry rates because they allow applications to further improve concurrency.

Independent Schemas

Linear transactions generalize well from operations across multiple keys to operations across multiple keys in different schemas. In our implementation, applications may create multiple schemas—which resemble tables from traditional database systems—and store different objects in each schema without any collisions in the key space. Clients construct the chain for transactions that touch multiple schemas by lexicographically ordering servers first by schema, then by key.

Nested Transactions

The architecture we have presented naturally supports nested transactions with only minimal changes to the client library. Nested transactions may be implemented by allowing transaction contexts to recursively refer to each other. Each nested transaction maintains its own locally-managed transaction context with a pointer to the parent transaction's context. Reads recursively query the parent context until either a cached value is read, or the root context issues the query to a storage server. Writes are stored in the transaction context to which they are issued. At commit time, the client merges a nested transaction into its parent context, by merging the read and write sets. Nested transactions abort if the values read in the child are modified in the parent or vice-versa. The client sends a linear transaction to the storage servers only when the root transaction commits.

4.3.2 Virtual Servers

Warp uses a system of virtual servers to map multiple partitions of the mapping to a single server. Clients construct their linear transaction chains by constructing a chain through the virtual servers, and then mapping these virtual servers to their respective servers. A server that maps to multiple virtual servers in a chain will appear at multiple places in the chain, where it acts as each of its virtual servers independently. Within each physical server, state is partitioned by virtual server, so that each virtual server functions as if it were independent. Virtual servers enable the system to perform dynamic load balancing more efficiently.

4.3.3 Coordinator

A replicated state machine called the coordinator partitions the key space across all data servers, ensures balanced key distribution, and facilitates membership changes as servers leave and join the cluster. Since the coordinator is not on the data path, its implementation is not critical to the performance of linear transactions.

The coordinator partitions data across servers and ensures balanced key distribution by using copyset replication [27] to group servers into replica sets. Each independent schema is partitioned across the generated copysets to create an object-to-server mapping. The coordinator over-partitions the key space to enable it to remap partitions from over-burdened replica sets to under-loaded replica sets if necessary.

As servers join and leave a cluster, the coordinator regenerates copysets to respond to new members. Servers dynamically compute the previous and next servers in each linear transaction's chain using the mapping; when the mapping changes, servers retransmit transactions whose chain changed. Every message carries the configuration's version to enable clients and servers to detect and reroute out-of-date requests using an up-to-date configuration. The mapping is changed incrementally, ensuring that each subsequent mapping overlaps with the previous mapping, which ensures that some replicas in each inlined chain will overlap as well. Thus, servers are always able to integrate new nodes without violating the assumptions used to construct linear transactions' chains.

The coordinator is implemented on top of the Replicant replicated state machine system. Replicant uses chain replication [121] to sequence the input to the state machine and a quorum-based protocol to reconfigure chains on failure. The details of Replicant are beyond the scope of this chapter; the function of the

coordinator could also be built on configuration services such as Chubby [22], ZooKeeper [58], and OpenReplica [5].

4.4 Evaluation

This section evaluates Warp's performance and scalability using the TPC-C benchmark The primary focus of thisevaluation is on examining the performance of Warp transactions relative to other transaction processing techniques. To that end, we implemented Sinfonia's mini-transactions [4] on top of HyperDex, hereafter referred to as MiniDex. Because Warp builds upon HyperDex, and because native HyperDex outperforms many NoSQL databases, we ensure a true apples-to-apples comparison by building all systems using the same code base. We also compare Warp to HyperDex, even though the latter offers no transactional guarantees. The client-facing interfaces and the benchmark code is identical for all three systems.

We performed our experiments on our dedicated lab-size cluster consisting of thirteen servers, each of which is equipped with two Intel Xeon $2.5\ \mathrm{GHz}$ E5420 processors, $16\ \mathrm{GB}$ of RAM, $500\ \mathrm{GB}$ SATA $3.0\ \mathrm{Gbit/s}$ hard disks, and Gigabit Ethernet. The servers are running 64-bit Ubuntu 14.04. Each storage system was configured with appropriate settings for a real deployment of this size. This includes setting the replication factor to be the minimum value necessary to tolerate one failure of any process or machine. Both the coordinators and the storage servers can each tolerate one failure. All systems provide strong consistency guarantees, which MiniDex and Warp extend across multiple objects.

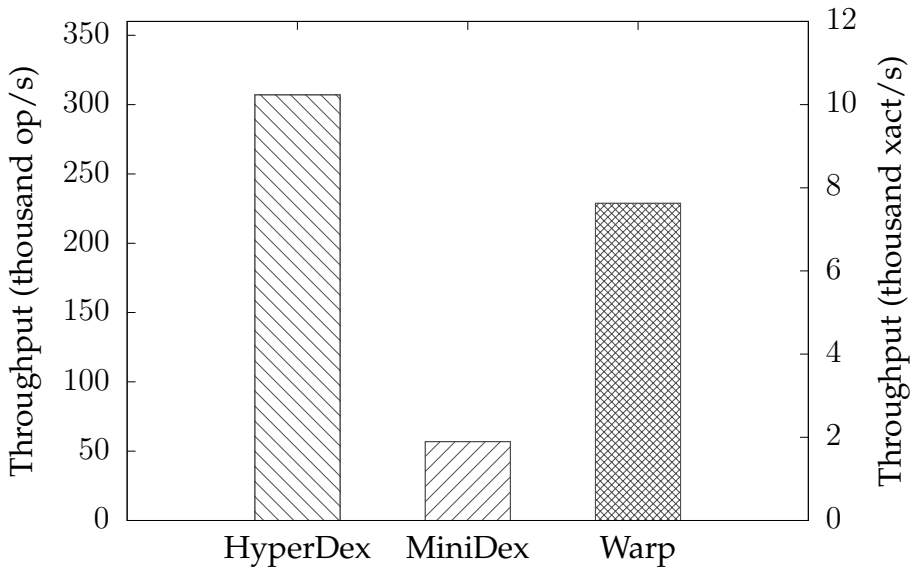


Figure 4.4: Total transactional throughput of the three systems. Warp outperforms MiniDex by a factor of 4, and achieves 75% the throughput of HyperDex, which Warp uses as its underlying key-value store. Warp averages approximately 7,500 transactions, or more than 225,000 individual key operations, per second in this benchmark.

4.4.1 TPC-C Macro Benchmark

The industry-standard TPC-C benchmark simulates an e-commerce application by specifying a mixed transaction workload. The workload specified by TPC-C is inherently difficult to process with optimistic concurrency control, because it includes both read-heavy and update-heavy transaction profiles and the update-heavy transactions intentionally contend on a small number of hot keys. For instance, the *new-order* transaction generates the order's identifier using a sequentially-increasing counter associated with one of one-hundred districts. The *payment* transaction increments the year-to-date totals for one of one-hundred districts and one of ten warehouses. The contention and interaction between the new-order and payment transaction profiles is what makes the TPC-C benchmark a compelling choice for testing new optimistic protocols.

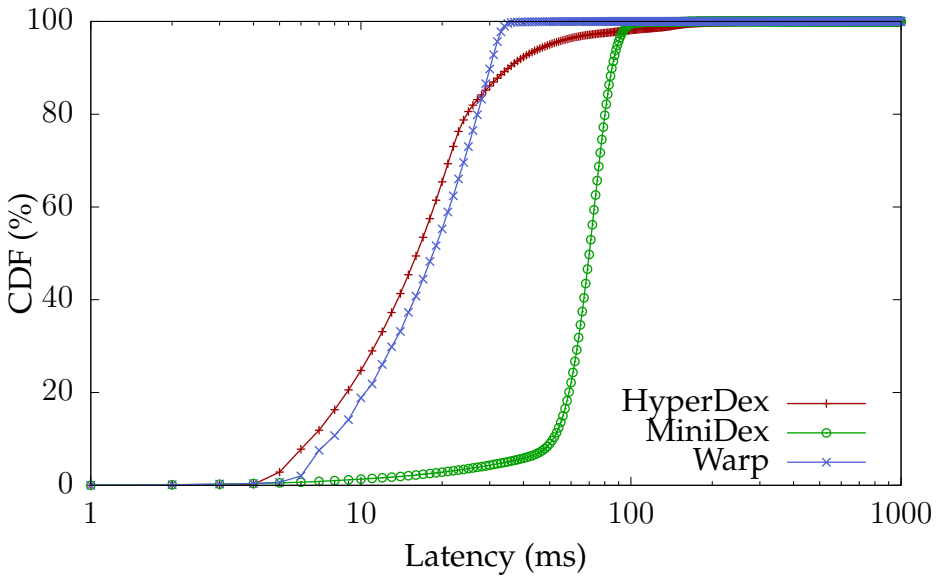


Figure 4.5: Latency of the TPC-C New Order transaction. HyperDex and Warp have similar latency profiles while MiniDex takes significantly longer to complete a transaction.

Profile	R	W	RMW	% Mix
New Order	12	3	11 (1)	45
Payment	0	1	3 (2)	45
Order Status	12	0	0	5
Stock Level	201 (1)	0	0	5

Table 4.1: A summary TPC-C workload. For each transaction profile, the chart shows the average number of read-only (R), write-only (W), and read-modify-write (RMW) operations. The TPC-C workload will randomly perform transactions accordingo to this distribution. The values in parenthesis specify the numberj of district or warehouse objects per transaction.

At the core of the benchmark are multiple transaction profiles which each represent a different type of application logic. Table 4.1 provides an overview of each transaction type. The values in parenthesis specify the number of district or warehouse objects per transaction. The bulk of the workload stems from the new-order and payment transaction profiles. These profiles simulate a customer placing purchase orders, and subsequently paying the invoice. Our implemen-

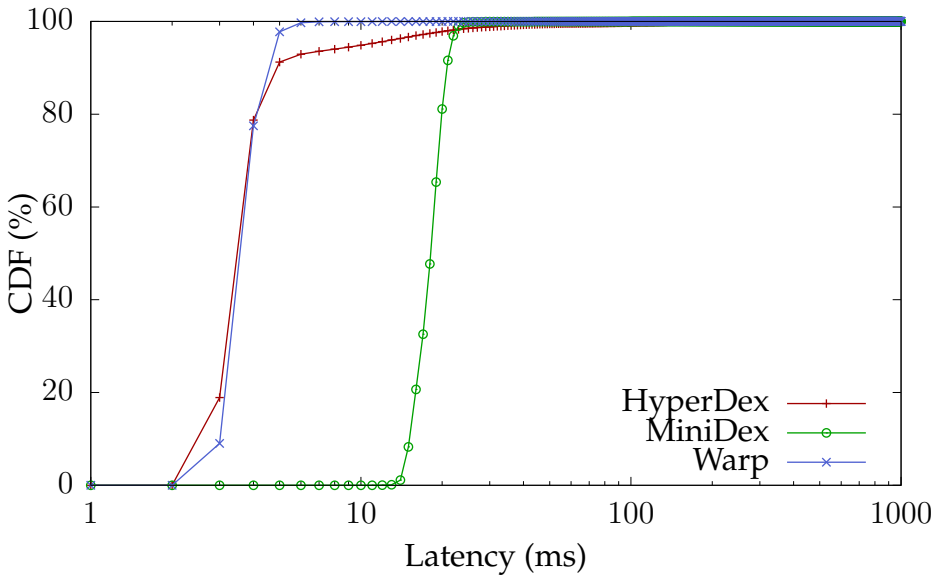


Figure 4.6: Latency of the TPC-C Payment transaction. HyperDex and Warp have similar latency profiles while MiniDex takes significantly longer to complete a transaction.

tation of TPC-C retains as much functionality of the benchmark as is reasonable to implement on a key-value store. In total, the implementation consists of approximately 1100 lines of Python code that execute client side using the Python bindings to HyperDex, MiniDex, and Warp. We omitted the "delivery transaction" profile because the TPC-C benchmark specifies that it be performed by a background process that would be handled by a messaging queue in a real deployment. Because we chose to retain most of the TPC-C benchmark's behavior, our results are incomparable to others in the literature that simply perform new-order transactions [13, 120].

We deployed the TPC-C benchmark with its default setting that includes 10 warehouses, which are very contended keys, and 100 districts, which are somewhat contended keys. Each new-order or payment transaction includes one warehouse and one district in the set of keys that it reads, modifies, and writes.

For the transactional systems, these keys will be the ones most likely to introduce transaction abort and retries. For the HyperDex workload, there cannot possibly be any conflict because the reads and writes may proceed in any order without transactional consistency. Intuitively, we expect that the performance of HyperDex provides an upper bound on throughput and a lower bound on latency for all experiments, because MiniDex and Warp add strictly more mechanism on top of the existing code. Consequently, the HyperDex upper bound allows us to objectively gauge how much overhead each system adds to the baseline.

Figure 4.4 shows the overall transactional throughput for HyperDex, MiniDex, and Warp. The experiment shows that that Warp achieves a throughput that is four times higher than MiniDex, and close to 7,500 transactions, or 225,000 operations, per second. To put the factor of 4 in perspective, Warp achieves 75% the throughput of the non-transactional system on which it builds, while MiniDex does not even realize 20% of its potential.

The intuition for why Warp is so much more efficient is two-fold: first, Warp's transaction management allows more concurrency than is possible with MiniDex; and second, Warp's communication costs are similar to those of the baseline and require no additional messages. Both systems construct chains to write data into the system, where each link in the chain equates to a network round trip. Where HyperDex will construct one chain of length $f + 1$ for each of the O operations, Warp will commit the operations through a single chain of length $O \times (f + 1)$ to commit the transaction. Thus, in the common case of no aborts and no retries, Warp requires no additional round trips beyond those required for a write within HyperDex. Figures 4.5 and 4.6 show latency CDFs for the new-order and payment transaction profiles. We can see that for both trans-

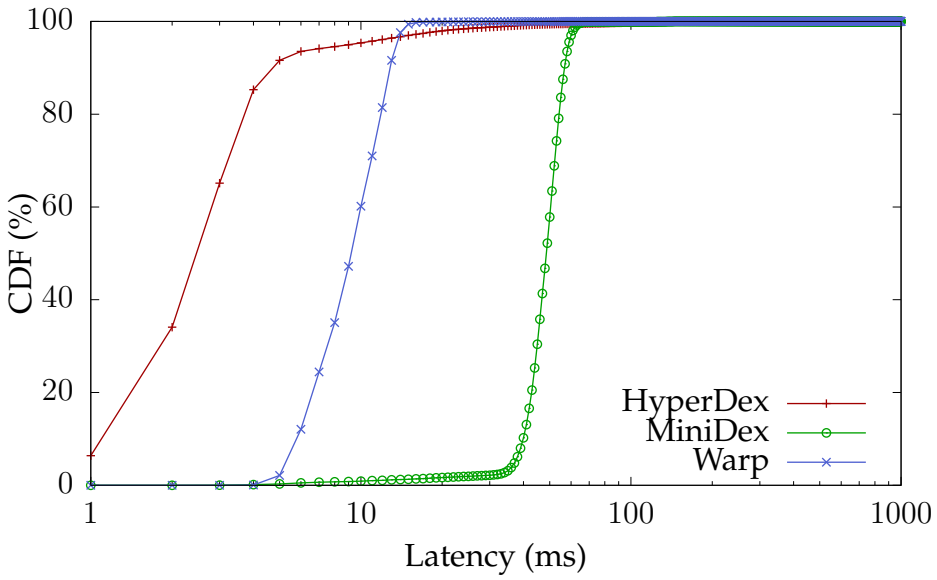


Figure 4.7: Read operations have different latencies inside and outside of a transaction. A non-transactional read may be directly performed against the server. A transactional read includes that latency, plus the cost of validating the read at commit time.

action types, the latency of HyperDex and Warp follow a similar trend, while the latency of MiniDex is approximately five times higher.

Because transactions must validate read operations, there's an additional cost to performing a transactional read that is not paid for non-transactional workloads. The read that the Warp client library performs to pull the object into the transaction context is the same cost as the read that the non-transactional code will perform. The Warp client then validates the read at commit time. In figure 4.7, we directly quantify the latency profile of the read-only "order status" transaction. We can see that Warp's latency is approximately three times higher than the non-transactional measurement, while the MiniDex latency is approximately six times higher.

Overall, the reason MiniDex achieves lower throughput and higher latency is because mini-transactions are more likely to abort. We observed, on average,

only 5% of transactions complete without aborting or retrying at least once, and we've included the time taken to retry transactions in the above numbers for all systems. Because all three systems use the same benchmark and baseline code, the performance difference is solely the commit protocol in use. MiniDex cannot permit multiple transactions to prepare for the same key simultaneously, forcing transactions to abort or wait, which increases the latency by a small constant multiplier. Warp permits these transactions to prepare simultaneously, enabling it to complete all transactions without aborting.

Although it may seem possible to relax the mini-transactions protocol to permit transactions to prepare for the same key simultaneously, doing so would break serializability. A modified MiniDex would require additional mechanisms to prevent the potential cycle illustrated in Figure 4.3, as concurrently prepared transactions could commit in different orders on different servers. Even HyperDex's atomic operations cannot enable such a relaxed commit protocol because they cannot affect the order in which operations occur on different servers.

4.5 Conclusion

This chapter describes Warp, a key-value store that provides serializable ACID transactions. The main insight behind Warp is a protocol called linear transactions which enables the system to completely distribute the task of ordering transactions. Consequently, transactions on separate servers will not require expensive coordination and the number of servers that process a transaction is independent of the number of servers in the system. The system achieves high performance on a variety of standard benchmarks, performing nearly as well as the non-transactional key-value store that Warp builds upon.

CHAPTER 5

GEO-REPLICATED TRANSACTIONS WITH CONSUS

5.1 Introduction

Geo-replication is a common feature among distributed storage systems. A geo-replicated system can withstand correlated failures up to and including entire data centers, and may reduce latency for clients by directing them to nearby data centers. These systems differ from systems designed for a single data center because they must account for latencies in the wide area that are orders of magnitude larger than the latency for communication in a single data center; a system designed for low latency settings will likely perform poorly if geo-distributed.

The latency between geographically distinct locations forces systems to navigate an inherent tradeoff between latency and fault tolerance. Systems may make an operation withstand a complete data center failure by incurring the latency cost to propagate it to other data centers before reporting that the operation has finished. On the other side of the tradeoff, systems may avoid the latency cost by reporting that an operation is complete before it propagates to other data centers—at the risk that the operation is lost in a failure and never takes effect. An optimal point in this tradeoff would uphold a desired fault tolerance guarantee while minimizing latency.

This chapter introduces Consus[1], a geo-replicated key-value store that supports strictly serializable cross-key transactions and executes transactions with three wide area message delays in the common case. It is easy to see why avoiding latency is desirable: any latency incurred on the critical path of a transaction directly impacts the performance of the application built on top. The decision

to uphold strong guarantees is more a matter of taste; in practice, organizations building on eventually consistent systems teach developers anti-patterns to avoid or build special-purpose storage systems for apps that are sensitive to consistency anomalies [76].

The core idea that enables Consus to reduce inter-data center latency is a commit protocol based upon generalized consensus [66]. Consus defers all inter-data center communication to the commit protocol—leaving the commit protocol to both globally replicate transactions and decide their outcomes. The commit protocol distributes the decision making process across all data centers and typically completes in three inter-data center message delays. Simply sending a message to a remote data center and receiving acknowledgement of its receipt—the bare minimum necessary to tolerate a failure—requires two such delays. Conventional commit protocols, such as 2-phase commit or Paxos, choose a single ensemble member to aggregate and disseminate information and communicating with this distinguished member necessarily incurs multiple inter-data center delays. Because Consus avoids a distinguished ensemble member, Consus commits with 33% less latency than other protocols, and incurs one more delay than the minimum necessary to uphold any degree of fault tolerance.

Consus' design makes a concerted effort to build on existing protocols—primarily Paxos—to provide a principled argument for the system's correctness. Simply reusing an existing implementation of multi-Paxos would suffice to uphold the safety guarantees of Consus and its commit protocol; however, we will see that a generic Paxos implementation introduces latency and failure sen-

[1]In Roman times, grain was essential to life and Consus served as the protector of grain; through its similarity to the word consilium, Consus became associated with secret conferences. In Modern times, Consus is very similar to the word consensus; not coincidentally the former uses the latter to serve as the protector of data.

sitivities that negatively impact performance. To overcome these limitations, Consus uses multiple optimized variants of Paxos that specialize Paxos to the task at hand, rather than treating it as a black-box component. Each Paxos implementation relies upon a different set of architectural constraints and protocol optimizations in order to decrease latency or improve availability without sacrificing the safety guarantees made by Paxos.

Overall, this chapter makes three contributions. First and foremost, this chapter presents a new commit protocol which reduces the inter-data center communication required to commit a transaction across multiple data centers. Second, it describes the optimized Paxos implementations within Consus outlining both the rationale behind each optimization and the way in which it differs from generic algorithms. Finally, this chapter describes the future direction of the project.

5.2 Design

Consus uses well-defined abstractions and builds upon proven protocols in order to constrain complexity. At the global scale, Consus treats each data center as a singular entity and runs a commit protocol across these entities. Internally each data center is not a singular entity, but a cluster providing its own partitioning and fault tolerance guarantees. Each cluster is further subdivided into a component for managing transaction execution, a component for storing the key-value pairs, and a component for executing the commit protocol. The commit protocol serves as the singular point for inter-data center communication and communicates with the local storage through a narrowly defined interface; consequently, it is agnostic to the internals of the transaction manager or key-value store. Figure 5.1 summarizes this architecture.

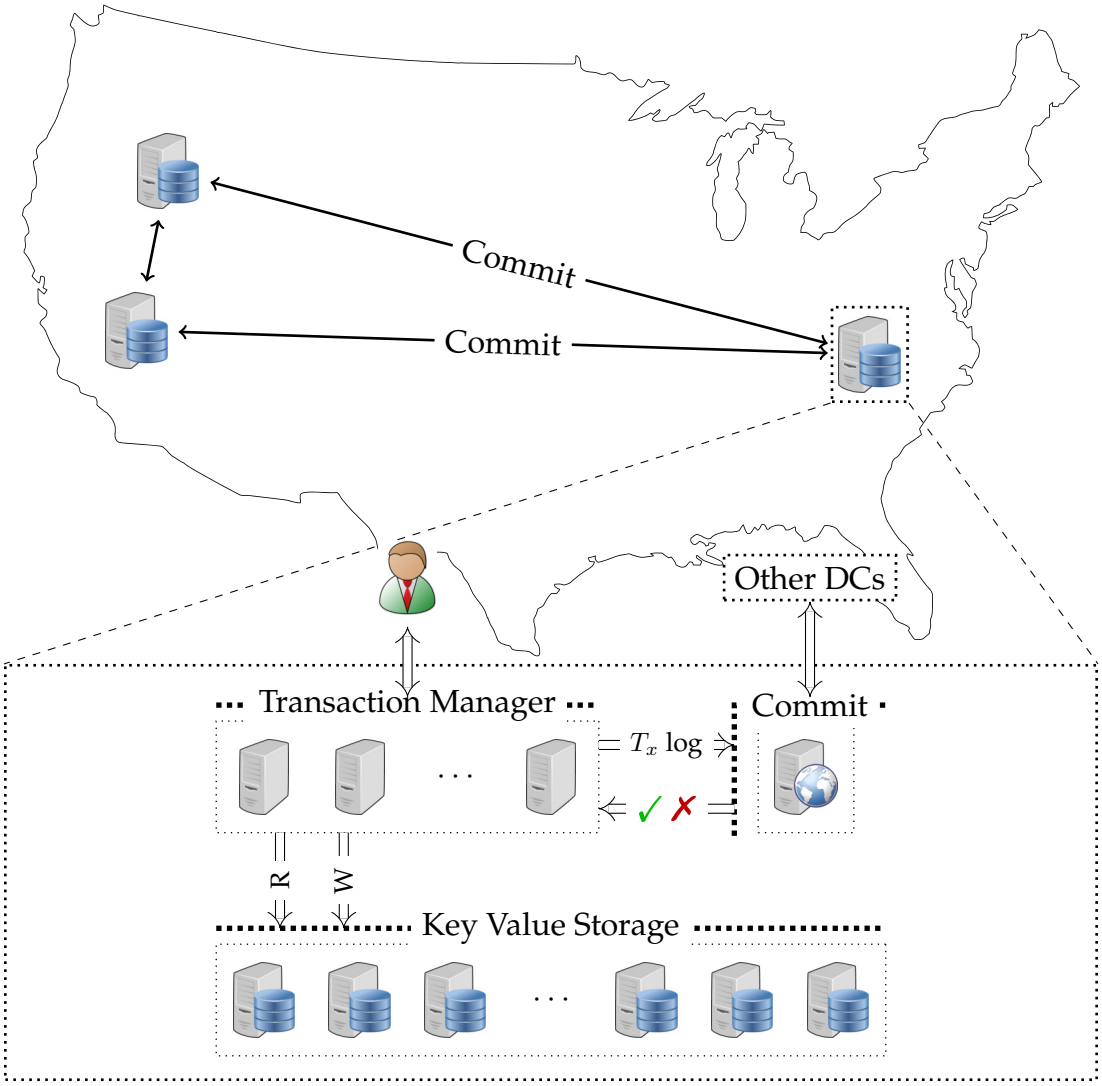


Figure 5.1: A typical Consus deployment spanning three data centers. Each data center resides in a single geographic region and is comprised of the commit protocol, a transaction manager, and key-value storage. The commit protocol serves as the sole method of communication between the data centers.

Consus' design has a realistic set of assumptions. It assumes that networks are asynchronous and that servers may experience crash or omission failures. Servers are assumed to recover all state from durable storage that was acknowledged as durable. Additionally, Consus assumes that there is some means by which servers that permanently fail will eventually be tagged as failed and

removed from the cluster in order to restore the fault tolerance guarantees of Paxos. This mechanism need not be timely or accurate; a slow detection leaves the cluster available, but with slightly weaker fault tolerance, while an erroneous detection will be treated like any other crash failure.

5.2.1 Commit Protocol

The Consus commit protocol handles the task of executing a transaction across multiple data centers. The protocol takes as input a transaction from one data center, replays it in other data centers, and outputs a decision to commit or abort the transaction. It is intentionally constrained to focus solely on committing or aborting a transaction leaving all transaction execution and concurrency control considerations to other components.

Consus decides a transaction's outcome in three logically distinct phases. In the first phase, a single data center executes the transaction; if the transaction executes to completion, it is sent to other data centers alongside sufficient information to determine that these data centers' executions match the execution in the original data center. In the second phase, each data center broadcasts the result of its own execution—whether it was able to reproduce the original execution—to all other data centers. In the final phase, the data centers feed these results to an instance of Generalized Paxos that allows all data centers to learn the transaction's outcome.

It is the information learned during the final phase, combined with the flexibility of Generalized Paxos, that enables the commit protocol's efficiency improvements. In 2-phase commit [51] and similar protocols [52, 109] the decision to commit is placed in an entity called a coordinator. Regardless of the coordinator's construction its fundamental purpose is to aggregate and disseminate

information between participants in the protocol. This necessarily introduces at least one message delay for the coordinator to learn any information, and at least two message delays before the other participants learn any information. As we shall see as we explore the full Consus commit protocol, all participants are able to learn the requisite information—and be assured that all other participants will learn the same information—in just one message delay during phase three.

Phase One

Phase one of the commit protocol executes a transaction in one data center and then re-executes it in the remaining data centers. A re-execution is deemed successful if and only if committing the transaction would leave the underlying data affected by the transaction in a state that is indistinguishable from the state of the same data in the initial data center. In this way, the re-execution preserves the original execution, and ensures all data centers present a consistent view of the data.

Re-execution may fail for a variety of environmental or workload reasons. For example, re-executions may diverge when concurrently executing transactions operate on the same data. The process executing a transaction informs the commit protocol when a re-execution diverges. This enables each data center participating in the commit protocol to use information about its own execution in subsequent stages.

At the end of phase one, a data center knows its own (re-)execution outcome, and, if the outcome is successful, is willing to hold the transaction ready to commit until the protocol finishes. The commit protocol requires that the transaction manager guarantee the transaction can commit if the protocol outputs a commit

decision.

Phase Two

Phase two of the commit protocol consists of each data center independently broadcasting the result of its phase one execution to all other data centers. It is not necessary for other data centers to finish—or have even started—executing a transaction before receiving the phase two message from another data center as long as the message's receipt is durably recorded.

There is no special insight to phase two because its purpose is exactly what it seems: Phase 2 informs most data centers about most other data center's executions. Because failure is inevitable, it is impossible for every data center to know about every other data center, nor whether any broadcasts were received. For this reason, while phase two is presented as a logically discrete action, even completely operational data centers will continually broadcast their execution to other data centers until the protocol runs to completion.

Phase Three

Phase three of the commit protocol enables data centers to combine the information broadcast by phase two into a single result learned by all data centers. Whereas phase two disseminates the outcome of the executions, phase three aggregates them in a durable manner and ensures that all data centers agree upon the aggregated value.

The value learned by data centers in phase three is a set of `commit` or `abort` results from the data centers involved in the transaction. This set of results is maintained by an instance of Generalized Paxos. As individual data centers learn the set of results from the Generalized Paxos protocol, the data centers

count the results and decide whether to commit or abort the transaction.

The Generalized Paxos protocol learns a partially ordered set (poset) of values. In phase three, these values are the phase two results, and there exists no order across these results. The primary contribution of Generalized Paxos is to provide a fast path by which acceptors can extend the accepted value poset by directly adding elements, so long as each acceptor has the same partial order across elements. Because phase two results are inherently unordered, acceptors can always propose these results and remain on the fast path of Generalized Paxos.

The structure of phase three allows each data center's acceptor to independently accept a value with sufficient information to decide to commit or abort. The data centers can then broadcast this accepted value with a single Paxos Phase 2B message to the other data centers. Once each data center receives a quorum of these Phase 2B messages, it can independently learn the poset of results without consulting other data centers.

In summary, at the beginning of phase three, each data center has the result from every other data center. Each data center proposes to its own local acceptor these values and then broadcasts its acceptor's state. This third broadcast is the third message delay in the commit protocol. Upon receiving the majority of these broadcasts, every data center can calculate the learned value of these accepted values without any single data center coordinating the learned value.

Avoiding Deadlock

The invariants that the Consus commit protocol imposes on the rest of the system make it possible for transactions to deadlock. Specifically, the constraint that a transaction remain committable until the commit protocol returns a de-

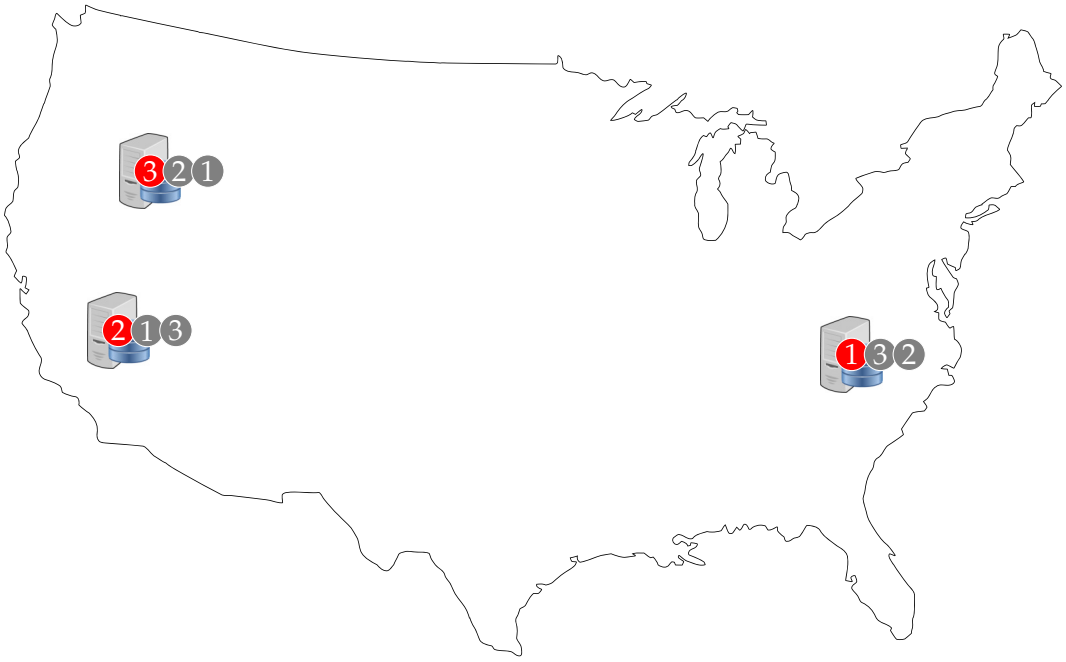


Figure 5.2: An example deadlock where three independent transactions block each other from making progress. The commit protocol can take upcalls from the transaction execution component to avoid deadlock.

cision will introduce behavior analogous the classic problem of deadlock in a lock-based database system. Figure 5.2 shows an example deadlock arising from three data centers each touching the same data.

To prevent deadlock, the commit protocol accepts upcalls from the transaction execution component that indicate when a transaction may be potentially deadlocked. Such an upcall signals to the commit protocol that the transaction might not ever commit and the protocol should act to avoid the deadlock. For some transactions, the transaction's outcome may already be decided by the time the upcall reaches the commit protocol. These transactions require no special consideration.

The commit protocol will attempt to abort transactions that have no outcome at the time of the upcall. This process is complicated by the fact that an upcall

may be generated within any data center, but all data centers must uniformly agree on a transaction's outcome. Consequently, all data centers must also agree to act to abort a transaction in response to a potential deadlock.

In response to deadlock, data centers may atomically signal their intent to retract a previously-recorded commit result and replace it with an abort result. To do this, the data center proposes the upcall to the Generalized Paxos instance of phase three of the commit protocol. Other data centers can then negate the previous `commit` result and count it as an `abort` result instead.

In order to ensure that a retraction is treated the same by all data centers, retractions are totally ordered with respect to all other elements in the learned poset. This partial ordering guarantees that data centers will not diverge when counting results. The data center tallies results in accordance with the partial order present in the set. If the commit total exceeds a quorum before a retraction is processed, the retraction is ignored; otherwise, the total shifts toward aborting the transaction. The partial order induced on the set ensures that results may be seen in any order, but the total `commit` and `abort` results encountered before a retraction are the same in all learned values. Similarly, because retractions are totally ordered all data centers will see the retractions in the same order.

The deadlock avoidance algorithm may force Generalized Paxos protocol to fall back to a classic Paxos round to resolve conflicts. These conflicts occur when the partial order at one acceptor differs from the partial order at another acceptor. Generalized Paxos will fall back to one or more rounds of the traditional Paxos protocol to resolve these conflicts. Because these additional rounds of Paxos take place in the wide area, they increase the number of message delays a transaction will encounter—however this cost is incurred only when data centers are attempting to abort a transaction due to deadlock.

Heuristics can be used to propose retractions to Generalized Paxos in a way that is unlikely to generate conflicts. For example, each data center may delay proposing any retractions to its local acceptor until the set of results it has previously proposed combined with the unproposed retractions would yield an abort outcome. The data center could then propose the retractions in a predetermined order that ensures that all data centers propose retractions in the same order. This is the simplest heuristic one could employ to keep Generalized Paxos from reverting to Classic Paxos; it is certainly worth investigating other heuristics alongside an investigation of ways to reduce the likelihood of abort upcalls in the underlying transaction execution engine.

Data Center Failure

Thus far, we have explored the commit protocol without regard to data center failure. While the protocol is resilient to a minority of data center failures, additional considerations are necessary to make the protocol return to a steady state after a data center recovers from failure. Specifically, any transactions executed while a data center is unavailable will not be propagated to the data center by the commit protocol; an additional mechanism is necessary to propagate any missed transactions.

A simple, but inefficient, approach is to have every data center continually synchronize state with other data centers. This ensures that any data committed in a majority of data centers will eventually propagate to all other data centers. The downside to this technique is that it requires background synchronization and any transaction that relies upon data that has yet to propagate will abort at all out-of-sync data centers.

In order to quickly bring data centers up to date and avoid background com-

munication, Consus also uses the information embedded in the phase one execution log to bring a data center up to date. The execution log includes information about every data item read during the transaction. Because transactions will only read committed data, it stands to reason that any read performed within a transaction is reading data that must exist in the key-value store. Consequently, the transaction execution engine can turn a read for missing data into an implicit write that restores missing data. While this is less efficient than reading the data, it incurs no latency waiting for the data center to become up-to-date—the data center can immediately commit the transaction.

Consus employs both of these techniques to recover from data center failure. The former technique ensures that all data becomes replicated to all requisite data centers, while the latter technique prevents a data center from appearing unavailable between cross-data center synchronization events.

5.2.2 Intra-Data Center Design

Within each data center, Consus is divided into two distinct execution components: a transaction manager and key-value storage. The transaction manager presents the sole interface to the client and uses two-phase locking as its concurrency control mechanism. The key-value storage serves as the component of record for each object stored within Consus and also stores the locks used by the transaction manager.

Transaction Manager

The transaction manager component durably records transactions during their execution. This ensures that all information about a transaction is recorded in one location and not scattered about the cluster. Upon failure of an entire data

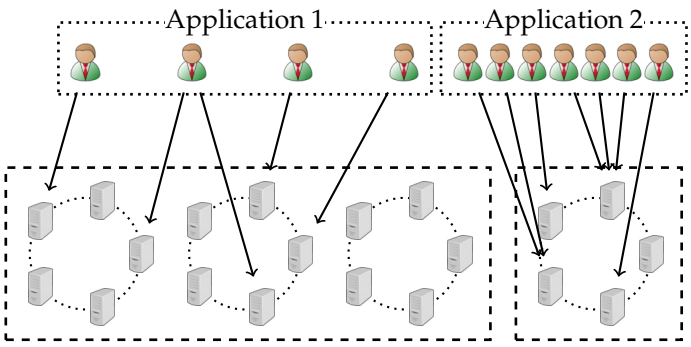


Figure 5.3: The transaction manager component. Transactions are sharded across Paxos groups that replicate each transaction as a state machine. Different applications may be directed to different state machines in order to provide performance isolation at the transaction level.

center, this logically centralized location provides a direct means to resume a transactions' execution without having to gather the information from many disparate points around the cluster.

For scalability and fault tolerance, the transaction manager is partitioned across multiple Paxos groups. Each transaction executes as a replicated state machine at exactly one of these Paxos groups. Consequently, the tier can be scaled to accommodate more transactions by adding additional servers and Paxos groups. With each additional Paxos group comes the ability to log more operations to disk. For workloads with low contention, this leads to a direct increase in performance. Figure 5.3 summarizes this architecture.

Transactions execute as a replicated state machine at a single Paxos group. Clients submit begin, read, write, and commit operations to the group, which then durably records and agrees upon the sequence of events issued by the client. The group interacts with the rest of the cluster on behalf of the client, acquiring locks, proxying reads, buffering writes, and releasing locks.

When the client commits the transaction, the Paxos group will hand the entire record of the transaction to the commit protocol. If the transaction commits,

the transaction manager will flush the buffered writes to the key-value store before releasing locks, while immediately acknowledging the commit to the client. Thus, even if the client fails, the transaction manager group may push a transaction to completion, or abort the transaction and clean up any state affected by the transaction.

One benefit of this sharded and replicated structure that is not immediately obvious is the ability to isolate multiple tenants' transaction managers by directing different tenants or applications to different Paxos groups. This isolation at the hardware level makes it impossible for one overly aggressive application to affect the performance of other, possibly higher priority, applications' transaction execution. This isolation only goes as far as isolation at the transaction manager level. Transactions that touch the same data are not isolated because of contention at the key-value store.

Key-Value Storage

The key-value store maintains a partitioned sorted map from bytestring keys to stored objects. It internally handles replication and partitioning of the data across multiple storage servers to enable the cluster to scale to many petabytes in size. By keeping the details of the key-value store's replication and partitioning mechanisms internal to the key-value component, the component's interface may be simplified to a small number of well-defined RPCs that may be issued to any server in the key-value store. This decision is in contrast to systems like HyperDex [41] that maintain logic in the client library for routing requests to servers, and re-routing or failing requests if the server configuration changes. Although the design in Consus potentially incurs an extra messaging hop within the data center, it avoids having to distribute the key-value map-

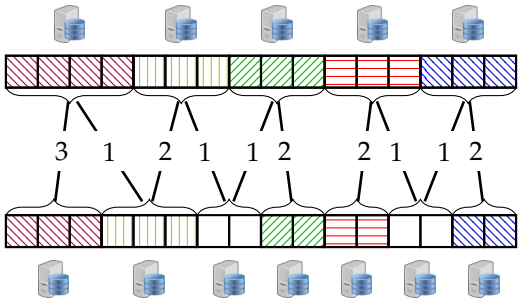


Figure 5.4: A key-value store mapping scaling from 5 nodes to 7. The overlap between the old and new mapping is used to calculate a preference matching servers to positions in the new mapping. The stable marriage algorithm then determines which ranges of partitions servers will adopt. Ranges left empty by the stable marriage algorithm are assigned to new servers.

ping to clients of the key-value store; this distribution can become increasingly expensive as the cluster grows in size, and the extra round trip avoids ever incurring this cost.

Consus divides the key-value space into a constant number of partitions in order to simplify the implementation and allow for global decisions during repartitioning events. By maintaining a constant number of partitions, Consus can compactly represent a mapping from each server to a range of partitions the server stores. Replicas of the data are stored by the servers adjacent to the data in the key space; this enables the servers to takeover serving a partition with minimal movement of data across servers. This design is in contrast to systems which store fixed-size ranges of the key-space and maintain a lookup table between keys and the range they map to. Such a scheme can grow to an unlimited number of constant-sized ranges and require that extra state be maintained to track the ranges themselves.

When servers join or depart the cluster, a single fault tolerant process within the cluster determines a globally optimal rebalanced form of the cluster and issues this new mapping to servers within the cluster. The process uses a variant

of the stable marriage algorithm [44] to rebalance the cluster. The algorithm creates a new ideal mapping to the key-value storage nodes that divides the constant number of partitions in the cluster into contiguous ranges of partitions, each of which will be assigned to a single node. It then assigns a preference each server has for each of these ranges by counting the number of partitions the server has that fall within the range. This preference between servers and partitions serves as input to the stable marriage algorithm, which will align each server with a range of partitions with a guarantee analogous to a stable marriage in the original algorithm. Figure 5.4 shows an 8-partition cluster growing from five nodes to seven, the preferences between the old nodes and new partitions, and how new nodes are distributed to the unassigned partitions.

To maximize the effectiveness of this assignment, the key-value store incrementally changes from one assignment to the next. Servers will incrementally adopt their assigned ranges, one partition at a time to ensure that at all times they are assigned to a contiguous range of partitions. This guarantees that any incremental assignment between two mappings can also serve as an input to the global optimization algorithm. Thus, work performed migrating from one configuration to the next will can be reused—even when the cluster dramatically changes in structure.

One open question regarding the stable marriage algorithm is the extent to which replication should be considered. The current design considers only a singular server for each partition, but could be adapted to consider replication when computing the weighting. In practice, this would likely produce a very similar result, but would be complex when disparate key spaces hosted on the key-value store specify different replication factors.

5.3 Paxos Optimizations

Consus uses multiple optimizations to Paxos to provide better performance than off-the-shelf Paxos implementations. In this section we will explore these optimizations and examine how they can improve on an off-the-shelf Paxos protocol. For each technique described, we will look at why the technique retains the safety that Consus requires of Paxos and, where relevant, why the technique is not generally applicable to all Paxos implementations.

5.3.1 Avoiding Paxos

The most elementary optimization is to avoid using Paxos entirely. The transaction state machine in the transaction manager was originally implemented using a standard Multi-Paxos protocol. Because of the unique structure of the transaction state machine, namely that there is a single source of all proposals, it was easy to completely avoid using Paxos with no loss of safety.

Because there is a single proposer, sequencing the operations does not require consensus—the single proposer has already sequenced the operations. The members of the ensemble need only durably log this sequence. In an off-the-shelf Multi-Paxos library, the single client would send operations through a single member that would propose the operations to the cluster.

The reason that Consus is able to avoid Paxos entirely in this particular instance is because the client and the transaction have a shared fate. If the client dies, there cannot be any more proposals issued to the cluster. Therefore when the client dies, the transaction log will not grow further. If a client dies, or takes too long to issue additional operations to a transaction, the transaction will be garbage collected and further operations will be rejected. This optimization can

only be applied when there exists a shared fate between the state machine and the source of proposed values.

5.3.2 Capturing Side Effects

The transaction state machine in Consus records the operations a client wishes to execute. These operations drive a state machine that acquires locks, performs reads, and checks that writes may succeed. Before a transaction may commit, these side effects must complete. The fact that operations are logged by the state machine does not imply that their side effects have completed.

In order to determine when a transaction has executed and may be passed to the commit protocol, Consus uses another application of consensus to achieve agreement among the state machine ensemble. Consus instantiates a new instance of the Paxos Synod protocol for each member of the state machine ensemble. These Synod instances use the ensemble as the set of acceptors. Each Synod instance captures the outcome of the transaction for a different member of the ensemble. Thus, the execution of a transaction—specifically its side effects—is captured in these Synod instances.

Once a quorum of these Synod instances capture a successful execution, the transaction can proceed with the global commit algorithm. This guarantees that the entire transactions log is recoverable so long as any quorum of the transaction ensemble remains live.

This structure is intended to prevent a case where each individual operation in the transaction log is replicated to a quorum of the ensemble, but no one server knows the entire log. Because the entire log is necessary to invoke the commit protocol, it is necessary to ensure that the log is not only durable to a minority of failures, but that the complete log is fully replicated on a majority

	S_1	S_2	S_3	S_4	S_5
`begin()`	✓	✓	✓		
`read(x)`	✓		✓	✓	
`write(x)`		✓		✓	✓
`commit()`	✓			✓	✓

Table 5.1: An example where every transaction operation is learned by a quorum of the servers (3), but no operation is learned by all servers, and no server learns of all operations.

of the ensemble. Table 5.1 shows a simple example of a log that is durably replicated, but that does not uphold the invariants necessary for Consus.

There is a somewhat obscure corner case in this configuration where a minority of servers become unreachable, but there is no majority that have confirmed having fully replicated the commit log. In practice, this is most likely to happen during unanticipated failures of the system; in theory, it could happen with an adversarial network. Accommodating this case has simplified the implementation of the transaction state machine and has caught multiple programming errors.

To overcome this problem case, the instances of the Synod protocol will accept a value that indicates that a server has failed. Servers may take up a ballot on their own or other servers' instances of the Synod protocol and mark that a server is failed. Invoking Synod instances in this manner ensures that all servers will only ever learn a `success` or a `failure` value for an instance of the Synod protocol The invariant upheld by Consus is that a server may propose any value for its own Synod instance, but may only propose `failure` for other servers. This allows any server to unilaterally decide that a data center aborts a transaction, but requires a majority of servers agree that the data center can commit the transaction.

The additional Synod instances also allow the transaction garbage collection

mechanism referred to earlier to safely abort local transactions. The garbage collector cannot directly propose to abort a transaction because the client is the only entity allowed to append operations to the transaction's log. Instead, the garbage collector can prevent the transaction from ever making progress by proposing `failure` to each Synod instance and running the protocol until they complete. Because this only ever happens on transactions that have expired, it will not delay their execution.

5.3.3 Implicit Phase 1 Paxos

Consus instantiates multiple instances of Paxos and the Paxos Synod protocol per transaction. This is in contrast to traditional uses of Paxos where there is one long-lived instance of Paxos. Consequently, Phase 1 of Paxos is invoked much more often than would otherwise happen. Because Phase 1 includes durably logging on remote machines, it can be expensive, and to invoke it multiple times per transaction adds unnecessary overhead.

Often, one member of a Paxos ensemble will be a suitable default leader for the Paxos group because it is through its actions that the group is created. In such an instance where the entity driving the Paxos group is known in advance, Consus implicitly sets the entity as having lead a successful Phase 1 ballot. This avoids the network latency associated with the proposer actually following Phase 1 of the protocol and the I/O cost of each acceptor durably recording its Phase 1 promise. When both are almost surely to succeed, actually executing the protocol along the regular path is wasteful. In the rare event that the first proposer for a Paxos instance is not leading the implicitly chosen ballot, the proposer could have thrashed with the likely proposer; the implicit Phase 1 delays the implicit proposer from thrashing because it will only learn of the

new ballot when one of its proposals is rejected by an acceptor.

This is purely a performance optimization that largely amounts to changing a variable's initialization. All members of the Paxos group must still retain the code paths necessary to perform Phase 1 of Paxos.

5.3.4 Recursive Generalized Paxos

The commit protocol discussed in Section 5.2.1 presents each data center as a singular entity participating in the protocol and running an instance of Generalized Paxos. Of course, if this were actually the case, a single server failure in one data center would make an entire data center appear offline, introducing more failure handling than would otherwise be desirable.

Consus makes the acceptors for the commit protocol's Generalized Paxos protocol fault tolerant by sequencing its inputs through a data center-local Generalized Paxos instance. Running the global protocol's acceptor on top of a local replicated state machine immediately makes the whole commit protocol withstand the failure of a single server without downtime; running it as a generalized state machine admits more concurrency and availability than would otherwise be available.

In a standard Paxos-replicated state machine, a single member of the ensemble operates as the leader for a ballot and proposes values using the authority of that ballot. If this member fails, another member of the ensemble must lead a higher ballot to continue proposing values to the cluster. This results in high latency during the transition, and deciding when a server is actually unavailable rather than executing slightly behind the others is a non-trivial task to do both quickly and without introducing thrashing between leaders.

In a Generalized Paxos-replicated state machine, any member of the ensem-

ble may propose any value by directly adding it to its local acceptor's partially ordered set (poset) of values. The value becomes accepted when the value appears in the greatest-upper-bound of the values accepted by a quorum of acceptors. Consequently, a value may be chosen once it reaches a quorum of the ensemble, without funneling the value through any single machine in the ensemble.

In a recursive Generalized Paxos-replicated state machine, an *inner state machine* runs on top of N ensembles running *outer state machines*, each of which simulates a single acceptor for the inner state machine. Messages that would normally be sent to a single acceptor in Generalized Paxos are wrapped in a proposal and sent to the outer state machine that simulates that acceptor. Likewise, messages that are normally sent to all acceptors are wrapped in proposals to all outer state machines. Each of these outer state machines learns a partially ordered set of messages that serve as the input to the acceptor that it represents for the inner state machine. Any messages generated by the inner state machine are sent to the requisite outer state machine ensembles.

The partially ordered set of messages in recursive Generalized Paxos permits a higher degree of concurrency than a total over messages would permit. By default any pair of messages will be ordered in the poset of messages unless a rule specifically exists that allows the messages to be unordered. The rules are:

- Phase 1B messages are always unordered with respect to other Phase 1B messages for the same ballot. These messages signal an acceptor following a new ballot and the order in which the acceptors follow the ballot is not significant.
- Phase 2B messages are unordered if there exists a greatest upper bound for the poset of the accepted values and the GUB is not $\perp$. In Consus each

of these values is a poset of results used in the commit protocol. These messages can commute because updating the acceptor's accepted value to the one contained in the Phase 2B message can only extend the learned value and cannot violate the safety invariants of Generalized Paxos.

- Proposals are ordered using the same ordering rules used for the relation over the values being proposed. In Consus, this means that proposed results may always commute, while proposed retractions will never commute.
- Proposals may commute with Phase 2B messages so long as the value being proposed is unordered with respect to every element in the accepted value contained within the Phase 2B message; again, this ordering uses the ordering relation over the inner state machine's poset elements.

These constraints are sufficient to enable recursive Generalized Paxos to efficiently decide the commit results without sending the results through any single machine for sequencing. The inner state machine is implicitly initialized to follow a ballot from the origin data center for a transaction. Then, absent a deadlock-triggered retraction, any number of Phase 2B and Proposal messages may arrive for the inner state machine and all will be unordered with respect to each other. At a high level of abstraction, each inner state machine is maintaining a set of `commit` or `abort` results and retractions for each data center. Each outer state machine maintains a copy of this inner state machine. At any instant in time the outer state machines may contain a different set of `commit` or `abort` results; however, the set will always be a subset what could be learned by a global observer who can see every message. A single retraction will force the outer state machines to converge on a single agreed-upon sequence to the inner state machine before continuing, thus forcing the inner state machine's

values to converge as well. It is an open question whether the ordering over the inner state machine's input could be further relaxed to admit more concurrency; it is certainly worth investigating more relaxed constraints alongside an investigation of ways to reduce the likelihood of abort upcalls in the transaction execution engine.

5.4 Implementation

The current Consus implementation is approximately $26\ \mathrm{k}$ lines of code that depends upon more than $44\ \mathrm{k}$ lines of supporting code that was originally written as dependencies of HyperDex. The Consus and HyperDex codebases are distinct. Because HyperDex was written with a different set of assumptions from Consus, with a different set of desiderata, the HyperDex implementation was not a useful starting point on which to build Consus.

The biggest difference between HyperDex and Consus is in its approach to fault tolerance. HyperDex makes an $f + 1$ fault tolerance assumption, where each unit of $f + 1$ nodes can withstand a concurrent failure of any f of those nodes. The implementation could not stand a concurrent failure of all $f + 1$ nodes without possibly losing data. Consus assumes that any or all nodes may fail and resume and its implementation has made this assumption from day one. Practically, this means that Consus is more conservative in its approach to data handling, opting to log data to disk rather than rely purely upon replication for fault tolerance. If Consus is to fluidly run in multiple data centers in the presence of failures, it must be able to run and restart in a single data center without incident, operator involvement, or a recovery procedure.

A more subtle result of the difference in fault tolerance assumptions is that Consus is engineered to hide latency anomalies to the extent possible. The Paxos

tricks in Section 5.3 outline some of the methods used to hide latency. The implementation also takes as much system coordination off the critical path as it can. In HyperDex a replicated coordinator maintains group membership for the system. This coordinator is on the critical path for failure recovery in HyperDex and must issue a new configuration after each failure to enable value-dependent chaining to route around the failure. Consus employs a similar replicated coordinator, but keeps the configuration out of the critical path for maintaining availability in the face of failures—Paxos provides higher availability under failure than HyperDex's chain-based protocol. The replicated coordinator is backed by Paxos, so ultimately both Consus and HyperDex remain available during a failure, but Consus does a better job of masking a failure and keeping latency consistent during a failure than HyperDex will.

5.5 Evaluation

In evaluating Consus, the chief question is whether the commit protocol provides the latency we would expect from the system. Theoretically, a transaction should commit in exactly three message delays plus twice the execution time of the transaction in a singular data center. To answer this question, we will look at Consus' performance in a single data center, extrapolate its expected performance to the multiple data center scenario, and then compare our expected results with the actual measurements.

The benchmarks in this section all use a version of the TPC-C benchmark adapted to a key-value store. The only changes from the standard TPC-C benchmark are that the data model was mapped onto a key-value store, doesn't use secondary indices for search, and does not have a workload running in the background.

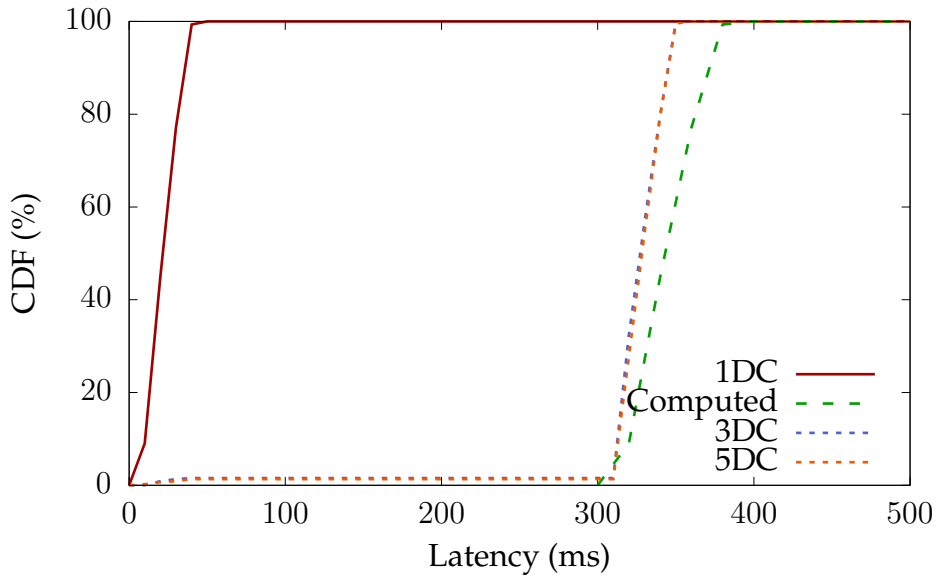


Figure 5.5: Consus commits new order transactions with predictable latency. In a single data center, transactions complete in less than 50 ms. In both 3 and five data centers, transactions commit in less than 400 ms

To simulate a five-data center deployment, the cluster runs on five Amazon m3.xlarge instances and the client runs on a single m3.xlarge instance, all within the same availability zone. Between the instances simulating the cluster, an artificial RTT of 200 ms was configured in order to provide a predictable 1-way latency of 100 ms. While this is not exactly the same as a system running in the wide area, the variance in latency within the local data center is orders of magnitude lower than the variance in the wide area latency, allowing for more accurate measurements of the latency.

The TPC-C benchmark was run three times, with one, three, and five simulated data centers respectively. For each configuration, the client offered the same load in a closed loop and measured the latency of each individual transaction. Our expectation, should Consus commit transactions in three message delays, should be that the latency of a transaction in three or five data centers is no more than twice the latency in a single data center plus 300 ms. Figure 5.5

shows that this is indeed the case.

5.6 Conclusion

This chapter introduces Consus, a new strictly serializable, geo-replicated key-value store. The key contribution of Consus is a commit protocol that can enable commit in three wide area message delays in the common case. Through the application of some Paxos optimizations, Consus is able to provide better theoretical performance than other strongly-consistent geo-replicated systems.

CHAPTER 6

TRANSACTIONAL FILESYSTEMS WITH WTF

6.1 Introduction

Distributed filesystems are a cornerstone of modern data processing applications. Key-value stores such as Google's BigTable [25] and Spanner [31], and Apache's HBase [12] use distributed filesystems for their underlying storage. MapReduce [36] uses a distributed filesystem to store the inputs, outputs, and intermediary processing steps for offline processing applications. Infrastructure such as Amazon's EBS [6] and Microsoft's Blizzard [81] use distributed filesystems to provide storage for virtual machines and cloud-oblivious applications.

Yet, current distributed filesystems exhibit a tension between retaining the familiar semantics of local filesystems and achieving high performance in the distributed setting. Often, designs will compromise consistency, require special hardware, or artificially restrict the filesystem interface. For example, in GFS, operations can be inconsistent or, "consistent, but undefined," even in the absence of failures [47]. GFS-backed applications must account for these anomalies, leading to additional work for application programmers. HDFS [9] side-steps this complexity by prohibiting concurrent or non-sequential modifications to files. This obviates the need to worry about nuances in filesystem behavior, but fails to support use cases requiring concurrency or random-access writes. Flat Datacenter Storage [87] is eventually consistent and requires a network with full-bisection bandwidth, which can be cost prohibitive and is not possible in all environments.

This chapter introduces the Warp Transactional Filesystem (WTF), a new distributed filesystem that exposes transactional support with a new API that pro-

vides *file slicing* operations. A WTF transaction may span multiple files and is fully general; applications can include calls such as read, write, and seek within their transaction. This file slicing API enables applications to efficiently read, write, and rearrange files without rewriting the underlying data. For example, applications may concatenate multiple files without reading them; garbage collect and compress a database without writing the data; and even sort the contents of record-oriented files without rewriting the files' contents.

The key design decision that enables WTF's advanced feature set is an architecture that represents filesystem data and metadata to ensure that filesystem-level transactions may be performed using, solely, transactional operations on metadata. Custom storage servers hold filesystem data and handle the bulk of I/O requests. These servers retain no information about the structure of the filesystem; instead, they treat all data as opaque, immutable, variable-length arrays of bytes, called *slices*. WTF stores references to these slices in HyperDex [42] alongside metadata that describes how to combine the slices to reconstruct files' contents. This structure enables bookkeeping to be done entirely at the metadata level, within the scope of HyperDex transactions.

Supporting this architecture is a custom concurrency control layer that decouples WTF transactions from the underlying HyperDex transactions. This layer ensures that transactions only abort when concurrently-executing transactions change the filesystem and generate an application-visible conflict. This seemingly minor functionality enables WTF to support concurrent operations with minimal abort-induced overheads.

Overall, this chapter makes three contributions. First, it describes a new API for filesystems called file slicing that enables efficient file transformations. Second, it describes an implementation of a transactional filesystem with minimal

overhead. Finally, it evaluates WTF and the file slicing interfaces, and compares them to the non-transactional HDFS filesystem.

6.2 Design

WTF's distributed architecture consists of four components: the metadata storage, the storage servers, the replicated coordinator, and the client library. Figure 6.1 summarizes this architecture. The metadata storage builds on top of HyperDex and its expansive API. The storage servers hold filesystem data, and are provisioned for high I/O workloads. A replicated coordinator service serves as a rendezvous point for all components of the system, and maintains the list of storage servers. The client library contains the majority of the functionality of the system, and is where WTF combines the metadata and data into a coherent filesystem.

In this section, we first explore the file slicing abstraction to understand how the different components contribute to the overall design. We will then look at the design of the storage servers to understand how the system stores the majority of the filesystem information. Finally, we discuss performance optimizations and additional functionality that make WTF practical, but are not essential to the core design, such as replication, fault tolerance, and garbage collection.

6.2.1 The File Slicing Abstraction

WTF represents a file as a sequence of byte arrays that, when overlaid, comprise the file's contents. The central abstraction is a *slice*, an immutable, byte-addressable, arbitrarily sized sequence of bytes. A file in WTF, then is a sequence of slices and their associated offsets. This representation has some inher-

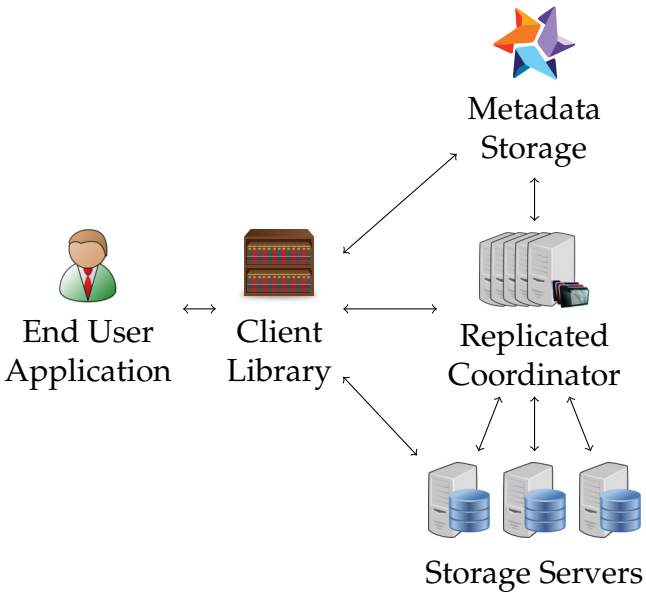


Figure 6.1: WTF employs a distributed architecture consisting of metadata storage, data storage, a replicated coordinator, and the client library. The client library unifies the metadata storage and storage servers to provide a filesystem interface.

ent advantages over block-based designs. Specifically, the abstraction provides a separation between metadata and data that enables filesystem-level transactions to be implemented using, solely, transactions over the metadata. Data is stored in the slices, while the metadata is a sequence of slices. WTF can transactionally change these sequences to change the files they represent, without rewriting data.

Concretely, file metadata consists of a list of *slice pointers* that indicate the exact location on the storage servers of each slice. A slice pointer is a tuple consisting of the unique identifier for the storage server holding the slice, the local filename containing the slice on that storage server, the offset of the slice within the file, and the length of the slice. Associated with each slice pointer is an integer offset that indicates where the slice should be overlaid when reconstructing the file. Crucially, this representation is self-contained: everything necessary to

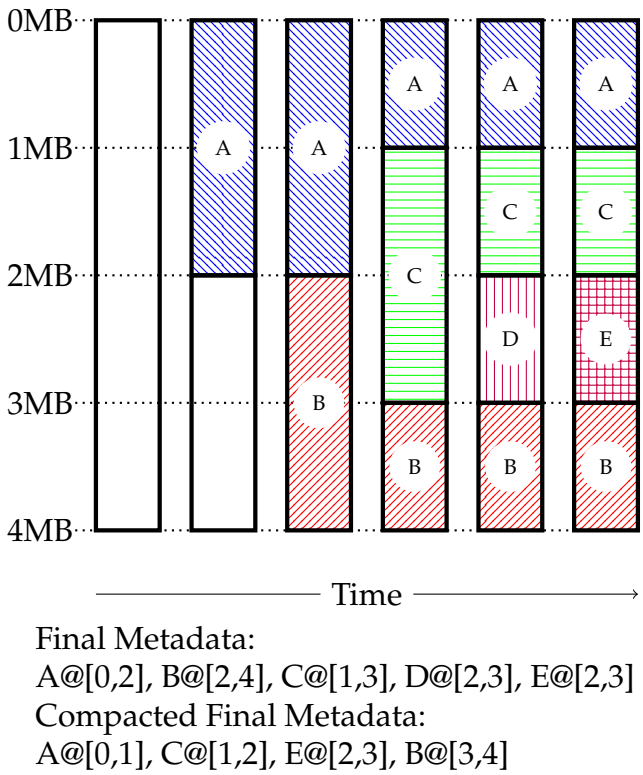


Figure 6.2: Writers append to the metadata list to change the file. Every prefix of the shown metadata list represents a valid state of the file at some point in time. The compacted metadata occupies less space by rearranging the metadata list to remove overwritten data.

retrieve the slice from the storage server is present in the slice pointer, with no need for extra bookkeeping elsewhere in the system. As we will discuss later, the metadata also contains standard info found in an inode, such as modification time, and file length.

This slice pointer representation enables WTF to easily generate new slice pointers that refer to subsequences of existing slices. Because the representation directly reflects the global location of a slice on disk, WTF may use simple arithmetic to create new slice pointers.

This representation also enables applications to modify a file with only localized modifications to the metadata. Figure 6.2 shows an example file consisting

of five different slices. Each slice is overlaid on top of previous slices. Where slices overlap, the latest additions to the metadata take precedence. For example, slice C takes precedence over slices A and B; similarly, slice E completely obscures slice D and part of C. The file, then, consists of the corresponding slices of A, C, E, and B. The figure also shows the *compacted* metadata for the same file. This compacted form contains the minimal slice pointers necessary to reconstruct the file without reading data that is hidden by another slice. Crucially, all file modifications can be performed by appending to the list of slice pointers.

The procedures for reading and writing follow directly from the abstraction. A writer creates one or more slices on the storage servers, and overlays them at the appropriate positions within the file by appending their slice pointers to the metadata list. Readers retrieve the metadata list, compact it, and determine which slices must be retrieved from the storage servers to fulfill the read.

The correctness of this design relies upon the metadata storage providing primitives to atomically read and append to the list. HyperDex natively supports both of these operations. Because each writer writes slices before appending to the metadata list, it is guaranteed that any transaction that can see these immutable slices is serialized *after* the writing transaction commits. It can then retrieve the slices directly. The transactional guarantees of WTF extend directly from this design as well: a WTF transaction will execute a single HyperDex transaction consisting of multiple append and retrieve operations.

6.2.2 Storage Server Interface

The file slicing abstraction greatly simplifies the design of the storage servers. Storage servers deal exclusively with slices, and are oblivious to files, offsets, or

concurrent writes. The minimal API required by file slicing consists of just two calls to create and retrieve slices.

A storage server processes a request to create a slice by writing the data to disk and returning a slice pointer to the caller. The structure of this request intentionally grants the storage server complete flexibility to store the slice anywhere it chooses because the slice pointer containing the slice's location is returned to the client only after the slice is written to disk. A storage server can retrieve slices by following the information in the slice pointer to open the named file, read the requisite number of bytes, and return them to the caller.

The direct nature of the slice pointer minimizes the bookkeeping required of the storage server implementation and permits a wide variety of implementation strategies. In the simplest strategy, which is the strategy used in the WTF implementation, each WTF storage server maintains a directory of slice-containing backing files and information about their own identities in the system. Each backing file is written sequentially as the storage server creates new slices.

As an optimization, each storage server maintains multiple backing files to which slices are appended. This serves three purposes: First, it allows servers to avoid contention when writing to the same file; second, it allows the storage server to spread data across multiple filesystems if configured to do so; and, finally, it allows the storage server to use hints provided by writers to improve locality on disk, as described in Section 6.2.7.

6.2.3 File Partitioning

Practically, it is desirable to keep the list of slice pointers small so that they can be stored, retrieved, and transmitted with low overhead; however, it would be

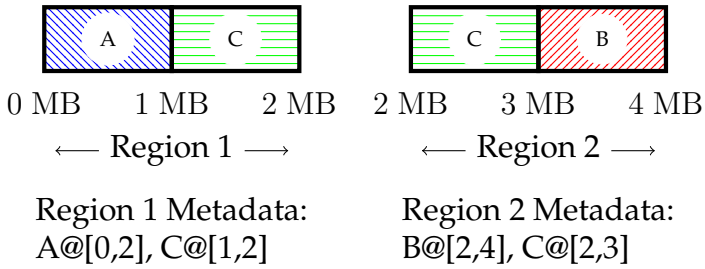


Figure 6.3: Files are partitioned into multiple regions to decouple the size of metadata lists from the size of the file. This figure shows the fourth state of the file from Figure 6.2 partitioned into 2 MB regions. Writes that are entirely within a single region are appended solely to that region's metadata. Writes that cross regions are transactionally appended to multiple lists.

API	Description
`yank(fd,sz):slice,[data]`	Copy `sz` bytes from `fd`; return slice pointers and optionally the data
`paste(fd, slice)`	Write `slice` to `fd` and increment the offset
`punch(fd, amount)`	Zero-out `amount` bytes at the fd offset, freeing the underlying storage
`append(fd, slice)`	Append `slice` to the end of file `fd`
`concat(sources, dest)`	Concatenate the listed files to create dest
`copy(source, dest)`	Copy source to dest using only the metadata

Table 6.1: WTF's new file slicing API. Note that these supplement the POSIX API, which includes calls for moving a file descriptor's offset via `seek`. `concat` and `copy` are provided for convenience and may be implemented with `yank` and `paste`.

impractical to achieve this by limiting the number of writes to a file. In order to achieve support for both arbitrarily large files and efficient operations on the list of slice pointers, WTF partitions a file into fixed size regions, each with its own list. Each region is stored as its own object in HyperDex under a deterministically derived key.

Operations on these partitioned metadata lists directly follow from the be-

havior of the system with a single metadata list. When an operation spans multiple regions, it is decomposed into one operation per region, and the decomposed operations execute within the context of a single HyperDex transaction. This guarantees that multi-region operations execute as one atomic action. Figure 6.3 shows a sample partitioning of a file, and how operations can span multiple metadata lists.

6.2.4 Filesystem Hierarchy

The WTF filesystem hierarchy is modeled after the traditional Unix filesystem, with directories and files. Each directory contains entries that are named links to other directories or files, and WTF enables files to be hard linked to multiple places in the filesystem hierarchy.

WTF implements a few changes to the traditional filesystem behavior to reduce the scope of a transaction when opening a file. Path traversal, as it is traditionally implemented, puts every directory along the path within the scope of a transaction, and requires multiple round trips to both HyperDex and the storage servers.

WTF avoids traversing the filesystem on open by maintaining a pathname to inode mapping. This enables a client to map a pathname to the corresponding inode with just one HyperDex lookup, no matter how deeply nested the pathname. To enable applications to enumerate the contents of a single directory, WTF maintains traditional-style directories, implemented as special files, alongside the one-lookup mapping. The two data structures are atomically updated using HyperDex transactions. This optimization simplifies the process of opening files, without significant loss of functionality.

Inodes are also stored in HyperDex, and contain standard information, such

as link count, modification time ,ownership, group, and permissions information, though WTF differs from POSIX in that permissions are not checked on the full pathname from the root. Each inode also stores a reference to the highest-offset region for the file, enabling applications to find the end of the file. The inode refers to a region instead of a particular offset so that the inode is only written when the file grows beyond the bounds of a region, instead of every time the file changes in size.

Because HyperDex permits transactions to span multiple keys across independent schemas, updates to the filesystem hierarchy remain consistent. For example, to create a hardlink for a file, WTF atomically creates a new pathname to inode mapping for the file, increments the inode's link count, and inserts the pathname and inode pair into the destination directory, which requires a write to the file holding the directory entries.

6.2.5 File Slicing Interface

The file slicing interface enables new applications to make more efficient use of the filesystem. Instead of operating on bytes and offsets as traditional POSIX systems do, this new API allows applications to manipulate subsequences of files at the structural level, without copying or reading the data itself.

Table 6.1 summarizes the new APIs that WTF provides to applications. The `yank`, `paste`, and `append` calls are analogous to read, write, and append, but operate on slices instead of sequences of bytes. The `yank` call retrieves slice pointers for a range of the file. An application may provide these slice pointers to a subsequent call to `paste` or `append` to write the data back to the filesystem, reusing the existing slices. These write operations bypass the storage servers and only incur costs at the metadata storage component.

The `append` call is internally optimized to improve throughput. A naive `append` call could be implemented as a transaction that seeks to the end of the file, and performs a `paste`. While not incorrect, such an implementation would prohibit concurrency because only one append could commit for each value for the end of file. Instead, WTF stores alongside the metadata list an offset representing the end of the region. An `append` call translates to a conditional list append call within HyperDex that only succeeds when the current offset plus the length of the slice to be appended does not exceed the bounds of the region. When an append is too large to fit within a single region, WTF will fall back on reading the offset of the end of file, and performing a write at that offset. This enables multiple `append` operations to proceed in parallel in the common case.

The remaining calls in the file slicing API are provided for convenience, as they may be implemented in terms of `yank` and `paste`. `concat` concatenates multiple files to create one unified output file. `copy` copies a file by copying the file's compacted metadata.

6.2.6 Transaction Retry

To guarantee that WTF transactions never spuriously abort, WTF implements its own concurrency control that retries aborted metadata transactions. WTF operations in the client library often read metadata during the course of an operation that is not exposed to the calling application. A change to this data after it is read may force the metadata transaction to abort, but to abort the corresponding WTF transaction would be spurious from the perspective of the application.

For example, consider a file opened in "append" mode. Each write to the

file must be written at the end-of-file offset, but the application does not learn this offset from the write. Internally, the client library computes the end of file, and then writes data at that offset. If the file changes in size between these two operations, the metadata transaction will abort. WTF masks this abort from the application by re-reading the end of file, and re-issuing the write at the new offset.

The mechanism that retries transactions is a thin layer between the WTF client library and the user's application. Each API call the application makes is logged in this layer by recording the arguments provided to the call and the value returned from the call. Should a metadata transaction abort during the WTF transaction commit, the WTF client library replays each operation from the log using the originally supplied arguments. If any replayed operation returns a value different from the logged call, the WTF transaction signals an abort to the application. Otherwise, WTF will commit the metadata changes from the replayed log to HyperDex. This process repeats as necessary until the metadata transaction, and, thus, the WTF transaction, commit, or a replayed operation triggers an WTF abort. This guarantees that WTF transactions are lockfree with zero spurious aborts.

To reduce the size of the replay log, the replay log refers to bytes of data that pass through the interface using slice pointers instead of copying the data. For example, a write of $100\ \mathrm{MB}$ will not be copied into the log; instead, the WTF client library writes the $100\ \mathrm{MB}$ to the requisite number of servers, and records the slice pointers in the log. Similarly, reads record slice pointers retrieved from the metadata, and not the slices themselves.

6.2.7 Locality-Aware Slice Placement

As an optimization, the WTF client library carefully places writes to the same region near each other on the storage servers to simultaneously improve locality for readers and to improve the efficiency of metadata compaction. When an application writes to a file sequentially, the locality-aware placement algorithm ensures that, with high probability, writes that appear consecutively in the metadata list will be consecutive on the storage servers' disks. During metadata compaction, the slice pointers for these consecutive writes are replaced by a single slice pointer that directly refers to the entire contiguous sequence of bytes on each storage server.

Two levels of consistent hashing [60] make it unlikely that two writes will map to the same backing files on the same storage server unless they are for the same metadata region. The WTF client library chooses the servers for each write by using consistent hashing across the list of storage servers. The client then provides the slice and identity of the metadata region to these servers, which use a different consistent hashing algorithm to map the write to disk. When collisions in the hash space do inevitably occur, it is unlikely that the colliding writes are issued so close in time as to be totally interleaved on disk in a way that eliminates opportunities for optimization.

6.2.8 Metadata Compaction and Defragmentation

The client library automatically compacts metadata during read and write operations to improve efficiency of future read and write operations. During write operations, the client library tracks the number of bytes written to both the metadata and the data for each region. When the ratio of metadata to data in a region exceeds a pre-defined threshold, the library retrieves the metadata list for

the region, compacts it as shown in Figure 6.2, and writes the newly compacted list. When reading, the client compacts the metadata list via the same process.

When metadata compaction alone cannot reduce the metadata to data ratio below the pre-defined threshold, the client library defragments the list by rewriting the data. The library rewrites fragmented data within a region into one single slice and replaces the metadata list with a single pointer to this slice. For efficiency's sake, defragmentation happens only on read, not on writes, because the client library necessarily reads the fragmented slices to fulfill the read; it can rewrite the slices without the overall system paying the cost of reading the fragmented slices twice. This mechanism is unused in the common case because locality-aware slice placement avoids fragmentation.

6.2.9 Garbage Collection

WTF employs a garbage collection mechanism to prevent the number of unreferenced slices from growing without bound. Metadata compaction and defragmentation ensures that metadata will not grow without bound, but in the process creates garbage slices that are not referenced from anywhere in the filesystem.

Because WTF performs all bookkeeping within the metadata storage, storage servers cannot directly know which portions of its local data are garbage. One possible way to inform the storage servers would be to maintain a reference count for each slice. This method, however, would require that the reference count on the storage server be maintained within the scope of the metadata transactions. Doing so, while not infeasible, would significantly complicate WTF's design and require custom transaction handling on the storage servers.

Instead of reference counting, WTF periodically scans the entire filesystem

metadata and constructs a list of in-use slice pointers for each storage server. For simplicity of implementation, these lists are stored in a reserved directory within the WTF filesystem so that they need not be maintained in memory or communicated out of band to the storage servers. Storage servers link the WTF client library and read the list of in-use slices to discover unused regions in their local storage space. The garbage collection mechanism runs periodically at a configurable interval that exceeds the longest-possible runtime of a transaction. Storage servers do not collect an unused slice until it appears in two or more consecutive scans.

Storage servers implement garbage collection by creating sparse files on the local disk. To compress a file containing garbage slices, a storage server rewrites the file, seeking past each unused slice. This creates a sparse file that occupies disk space proportional to the in-use slices it contains. Files with the most garbage are the most efficient to collect, because the garbage collection thread seeks past large regions of garbage and only writes the small number of remaining slices. Backing files with little garbage incur much more I/O, because there are more in-use slices to rewrite. WTF chooses the file with the most garbage to compact first, because it will simultaneously delete the most garbage and incur the least I/O. Some filesystems enable applications to selectively punch holes in the file without rewriting the data; although our implementation does not use these capabilities, an improved implementation could do so.

6.2.10 Fault Tolerance

WTF uses replication to add fault tolerance to the system. Changing WTF to be fault tolerant requires modifying the metadata lists' structure so that each entry references multiple replicas of the same data, each with a different slice pointer.

On the write path, writers create multiple replica slices on distinct servers and append their pointers atomically as one list entry. Readers may read from any replica, as they hold identical data.

The metadata storage derives its fault tolerance from the guarantees offered by HyperDex. Specifically, that it can tolerate f concurrent failures for a user-configurable value of f. HyperDex uses value-dependent chaining to coordinate between the replicas and manage recovery from failures [41].

6.3 Implementation

Our implementation of WTF implements the file slicing abstraction. The implementation is approximately $30\ \mathrm{k}$ lines of code written. It relies upon HyperDex with transactions, which is approximately $85\ \mathrm{k}$ lines of code, with an additional $37\ \mathrm{k}$ lines of code of supporting libraries written for both projects. The replicated coordinator for both HyperDex and WTF is an additional $19\ \mathrm{k}$ lines of code. Altogether, WTF constitutes $171\ \mathrm{k}$ lines of code that were written for WTF or HyperDex.

WTF's fault tolerant coordinator maintains the list of storage servers and a pointer to the HyperDex cluster. It is implemented as a replicated object on top of Replicant, a Paxos-based replicated state machine service. The coordinator consists of just 960 lines of code that are compiled into a dynamically linked library that is passed to Replicant. Replicant deploys multiple copies of the library, and sequences function calls into the library.

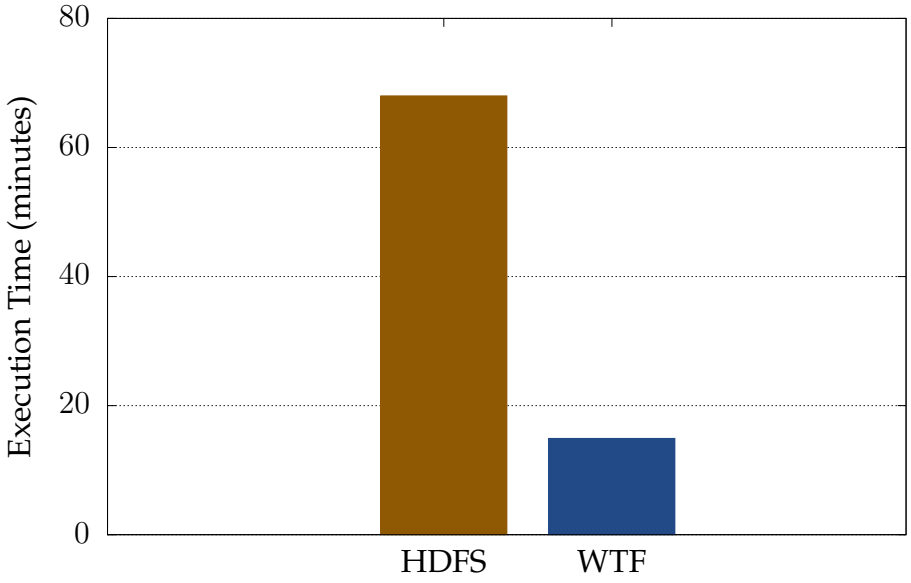


Figure 6.4: Total execution time for sorting a 100 GB file using a map-reduce application. HDFS takes more than one hour and seven minutes to sort the file, while WTF completes the same task in under fifteen minutes.

6.4 Evaluation

To evaluate WTF, we will look at a series of both end-to-end and micro benchmarks that demonstrate WTF under a variety of conditions. The first part of this section looks at how the features of WTF may be used to implement a variety of end-to-end applications. We will then look at a series of microbenchmarks that characterize the performance of WTF's conventional filesystem interface.

All benchmarks execute on a cluster of fifteen dedicated servers. Each server is equipped with two Intel Xeon 2.5 GHz L5420 processors, 16 GB of DDR2 memory with ECC, and between 500 GB and 1 TB SATA spinning-disks. The servers are connected with gigabit ethernet via a single top of rack switch. Installed on each server is 64-bit Ubuntu 14.04, HDFS from Apache Hadoop 2.7, and WTF with HyperDex.

For all benchmarks, HDFS and WTF are configured similarly. Both systems

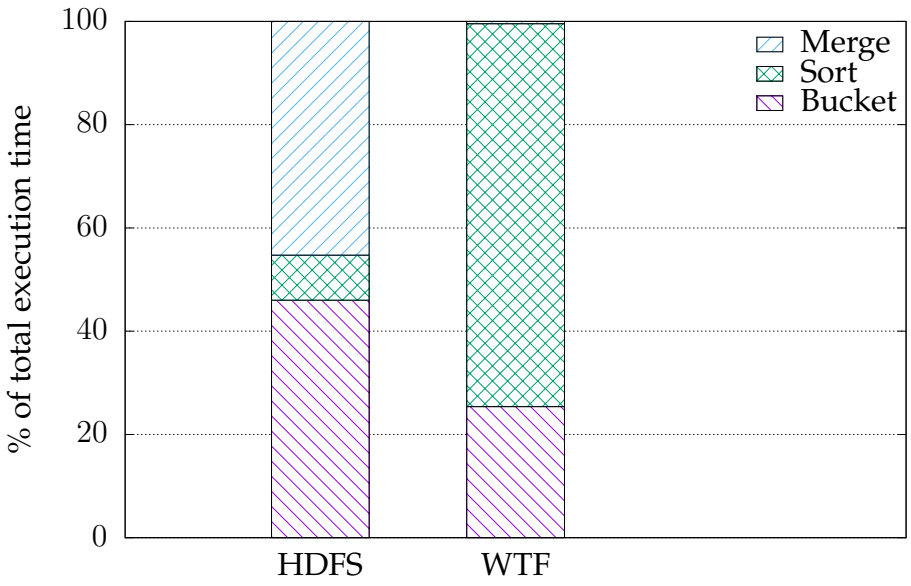


Figure 6.5: Execution time of the sort broken down by stage of the map-reduce application. HDFS spends 91.5% of its time partitioning and reassembling the data, compared to WTF, which spends 25.9% of its time on the same task.

are deployed with three nodes reserved for the meta-data—a single HDFS name node, or a HyperDex cluster—and the remaining twelve servers are allocated as storage nodes for the data. Clients are spread across the twelve storage nodes. Except for changes necessary to achieve feature parity, both systems were deployed in their default configuration. To bring the semantics of HDFS up to par with WTF, each `write` is followed by an `hflush` call to ensure that the write is flushed from the client-side buffer to HDFS. The `hflush` ensures that writes are visible to readers, and does *not* flush to disk. This is analogous to changing from the C library's `fwrite` to a UNIX `write` in a traditional application. The resulting guarantees are equivalent to those provided by WTF.

Additionally, in order to work around a bug with append operations [10], the HDFS block size was set to $64\ \mathrm{MB}$. Without this change to the configuration, HDFS can report an out-of-disk-space condition when only 3% of the disk space

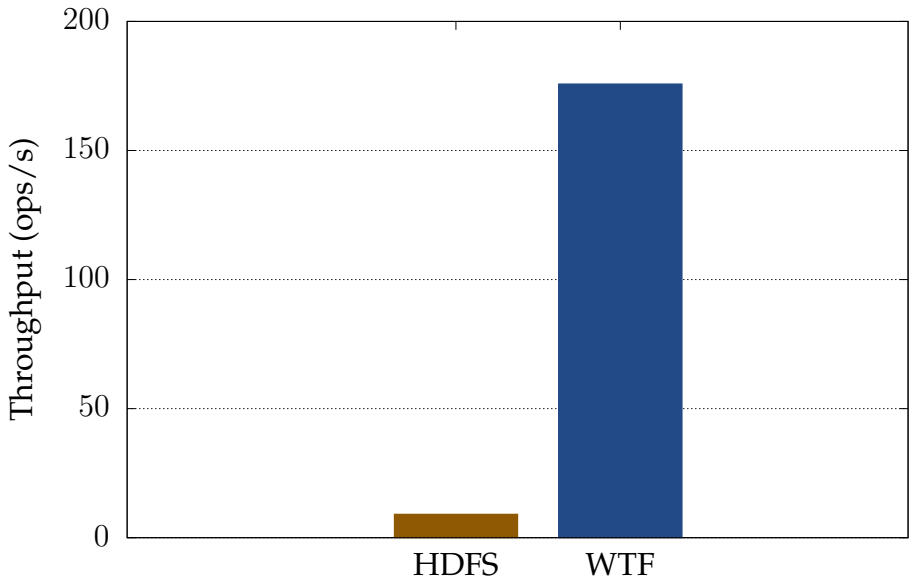


Figure 6.6: A concurrent work queue implemented on top of a distributed filesystem. The implementation based on WTF achieves throughput $19\times$ that of HDFS.

is in use. Instead of gracefully handling the condition and falling back to other replicas as is done in WTF, the failure cascades and causes multiple writes to fail, making it impossible to complete some benchmarks. The change is unlikely to impact the performance of data nodes because the increase from 64 MB to 128 MB was not motivated by performance [11]. WTF is also configured to use 64 MB regions.

Except where otherwise noted, both systems replicate all files such that two copies of the file exist. This allows the filesystem to tolerate the failure of any one storage server throughout the experiment without loss of data or availability. It is possible to tolerate more failures so long as all the replicas for a file do not fail simultaneously.

Stage	Conventional	File Slicing
Bucketing	R = 100 GB	R = 100 GB
	W = 100 GB	W = 0 GB
Sorting	R = 100 GB	R = 100 GB
	W = 100 GB	W = 0 GB
Merging	R = 100 GB	R = 0 GB
	W = 100 GB	W = 0 GB
Total	R = 300 GB	R = 200 GB
	W = 300 GB	W = 0 GB

Table 6.2: File slicing reduces the amount of data written to the filesystem by one third compared to conventional APIs.

6.4.1 Applications

This section examines multiple applications that each demonstrate a different aspect of WTF's feature set.

Map Reduce: Sorting MapReduce [36] applications often build on top of filesystems like HDFS and GFS. In MapReduce, sorting a file is a three-step process that breaks the sort into two map jobs followed by a reduce job. The first map task partitions the input file into buckets, each of which holds a disjoint, contiguous section of the keyspace. These buckets are sorted in parallel by the second map task. Finally, the reduce phase concatenates the sorted buckets to produce the sorted output.

Each intermediate step of this application is written to the filesystem and the entire data set will be read or written several times over. Here, WTF's file slicing API can improve the efficiency of the application by reducing this excessive I/O. Instead of reading and writing whole records, WTF-based sort uses `yank` and `paste` to rearrange records. File slicing eliminates almost all I/O of the reduce phase using a `concat` operation.

Empirically, file slicing operations improve the running time of WTF-based

sort. Figure 6.4 shows the total running time of both systems to sort a $100\ \mathrm{GB}$ file consisting of $500\ \mathrm{kB}$ records indexed by $10\ \mathrm{B}$ keys that were generated uniformly at random. In this benchmark, the intermediate files are written without replication because they may easily be recomputed from the input. We can see that WTF sorts the entire file in one fourth the time taken to perform the same task on HDFS.

The speedup is largely attributable file-slicing. From Figure 6.5, we can see that the WTF-based sorting application spends less time in the partitioning and merging steps than the HDFS-based sort. HDFS spends the majority of its execution time performing I/O tasks; just 8.5% of execution time is spent in the CPU-intensive sort task. In contrast, WTF spends 74.1% of its time in the CPU intensive task and seconds in the merge task.

Work Queue Work queues are a common component of large scale applications. Large work units may be durably written to the queue and handled by the application at a later point in time in FIFO order.

One simple implementation of a work queue is to use an append-only file as the queue itself. The application appends each work unit to the file, and can dequeue from the work queue by reading through the file sequentially—the file itself encodes the FIFO nature of the queue. This benchmark consists of an application with multiple writers that concurrently write to a single file on the filesystem. Each work unit is $1\ \mathrm{MB}$ in size and written atomically. The application runs on each client server, for a total of twelve application instances.

Figure 6.6 shows the aggregate throughput for the work queue built on top of both HDFS and WTF. We can see that WTF's throughput is $19\times$ that of HDFS for this workload. Each work unit is saved to WTF in $55\ \mathrm{ms}$, while the application built on HDFS waits $1.3\ \mathrm{s}$ on average to enqueue each work unit.

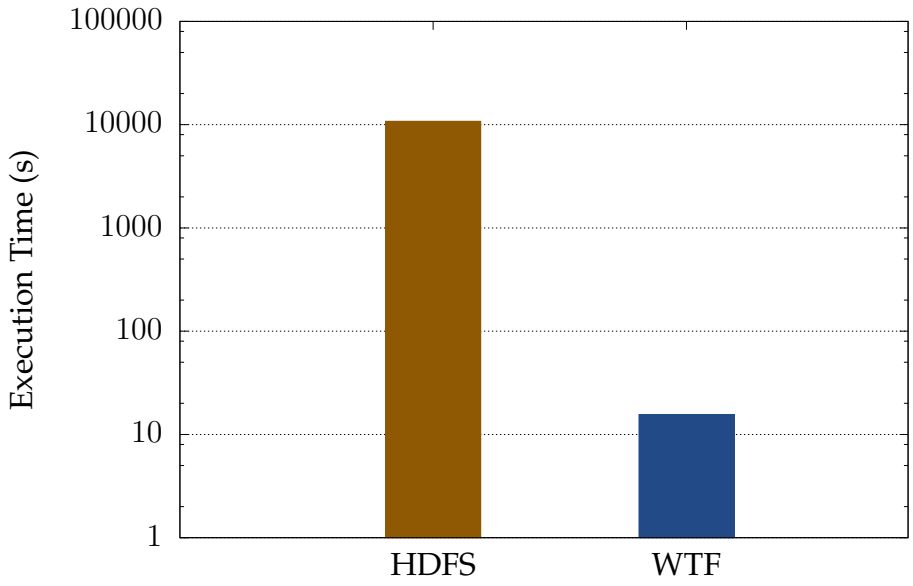


Figure 6.7: Time taken to generate a readily-playable video file from individual scenes.

Image Host Image hosting sites, such as flickr or imgur have become the de-facto way of sharing images on the Internet. While imgur's implementation serves images from Amazon S3, Facebook's image serving solution, called Haystack [17], stores multiple photos in a single file to reduce the costs of reading and maintaining metadata. In Haystack, servers read into memory a map of the photos' locations on disk so that reading a single photo from disk does not entail any additional disk reads.

This example application models an imgur-like site built using the multi-photo file technique used within Haystack. Photos are written to multi-gigabyte files, each of which has a footer mapping photos to their offsets in the files. The application loads this map into memory so that it may serve requests by locating the file's offset within the map, and then reading the file directly from the offset in the filesystem.

To better simulate a real photo-sharing website, photos are randomly gener-

ated to match the size of photos served by imgur. The distribution of photo sizes was collected from the front page of imgur.com over a 24-hour period. Because imgur serves both static images and gifs, the size of photos varies widely. Median image size is $332\ \mathrm{kB}$, while the average image size is $8.5\ \mathrm{MB}$. Because the precise request distribution of requests is not available from imgur, the workload re-uses the Zipf request distribution specified for YCSB workloads [30]. For this workload, we measured that WTF achieves 88.8% the throughput of the same application on top of HDFS. The performance difference is largely attributable to the reads of smaller files. As we will explore in the microbenchmarks section, WTF needs further optimization for small read and write operations.

Video Editing WTF's file slicing API can be used to reorganize large files with orders of magnitude less I/O. One particular domain where this can be useful is video editing of high-definition raw video. Such videos tend to be large in size, and will be rearranged frequently during the editing process. While specialized applications can edit and then play back videos, WTF enables another point in the design space.

This application uses WTF's file slicing to move scenes around in a video file without physically rewriting the video. The chief benefit of this design, over editors on existing filesystems, is that an off-the-shelf video player can play the edited video file because it is in a standard container format. To benchmark this application, we used our video editor to randomly rearrange the scenes in a $2\ \mathrm{h}$ movie, such that the movie out of chronological order. The source material was 1080p raw video dumped from a Bluray disk. Overall the raw video/audio occupies approximately $377\ \mathrm{GB}$ or $52\ \mathrm{MB/s}$. Figure 6.7 shows the time taken to rewrite the file using HDFS's conventional API compared to WTF's file-slicing

```
# wtf fuse ./mnt
# cd ./mnt
# wtf fuse-begin-transaction
# ls
/data.0000  /data.0001
/data.0002  /data.0003
....
# rm *
# ls
# wtf fuse-abort-transaction
# ls
/data.0000  /data.0001
/data.0002  /data.0003
....
```

Figure 6.8: WTF's transactional functionality enables users to manipulate the filesystem in isolation.

API. WTF takes three orders of magnitude less time to make a file readable—on the order of seconds—while conventional techniques require nearly three hours.

Sandboxing The transactional API of WTF makes it easy to use the filesystem as a sandbox where tasks may be committed or aborted depending on their outcome. The WTF implementation includes a FUSE module that enables users to mount the filesystem as if it were a local filesystem. This enables shell navigation of the filesystem hierarchy and allows regular applications to read and write WTF filesystems without modification. In addition to implementing the full filesystem interface, the FUSE bridge exposes special `ioctl`s to permit users to control transactions. Users may begin, abort, or commit transactions via command-line tools that wrap these `ioctl`s.

The transactional features of the FUSE bridge enables users to perform risky actions within the context of a transaction; the transactional isolation provides a degree of safety users would otherwise not be afforded. The actions taken by

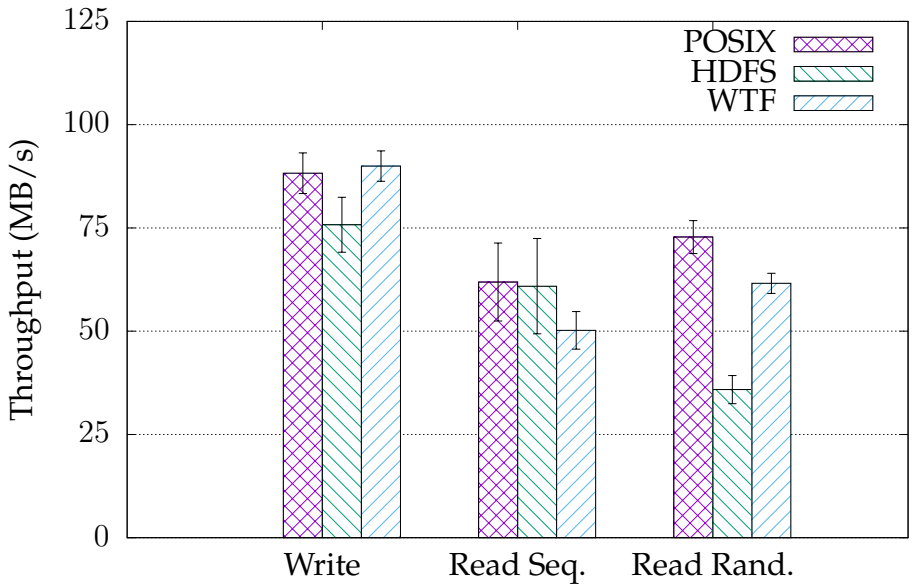


Figure 6.9: Performance of a one-server deployment of HDFS and WTF compared with the ext4 filesystem. Error bars indicate the standard error of the mean across seven trials.

the user are not visible until the user commits, and should the user abort, the actions will never be persisted to the filesystem. Figure 6.8 shows a sample interaction with an WTF filesystem containing data for a sample research project. We can see that the user begins a transaction and inadvertently removes all of the research data. Because the errant `rm` command happened in a transaction, the data remains untouched.

6.4.2 Micro Benchmarks

In this section we examine a series of microbenchmarks that quantify the performance of the POSIX API for both HDFS and WTF. Here HDFS serves as a gold-standard. With ten years of active development, and deployment across hundreds of nodes, including deployments at both Facebook and LinkedIn [27], HDFS provides a reasonable estimate of distributed filesystem performance. Al-

though we cannot expect WTF to grossly outperform HDFS—both systems are limited by the speed of the hard disks in the cluster—we can use the degree to which WTF and HDFS differ in performance to estimate the overheads present in WTF's design.

Setup The workload for these benchmarks is generated by twelve distinct clients, one per storage server in the cluster, that all work in parallel. This configuration was chosen after experimentation because additional clients do not significantly increase the throughput, but do increase the latency significantly. All benchmarks operate on $100\ \mathrm{GB}$ of data, or over $16\ \mathrm{GB}$ per machine once replication is accounted for. This is large enough that our workload blocks on disk on Linux [72].

Single server performance This first benchmark executes on a single server to establish the baseline performance of a one node cluster. Here, we'll compare the two systems to each other and the same workload implemented on a local ext4 filesystem. The comparison to a local filesystem provides an upper bound on performance. To reduce the impact of round trip time in each distributed system the client and storage server are collocated. Figure 6.9 shows the throughput of write and read operations in the one-server cluster. From this we can see that the maximum throughput of a single node is $87\ \mathrm{MB/s}$, which means the total throughput of the cluster peaks at approximately $1\ \mathrm{GB/s}$.

Sequential Writes WTF guarantees that all readers in the filesystem see a write upon its completion. This benchmark examines the impact that write size has on the aggregate throughput achievable for filesystem-based applica-

[0]Blocks $<256\ \mathrm{kB}$ wrote smaller files to limit execution time.

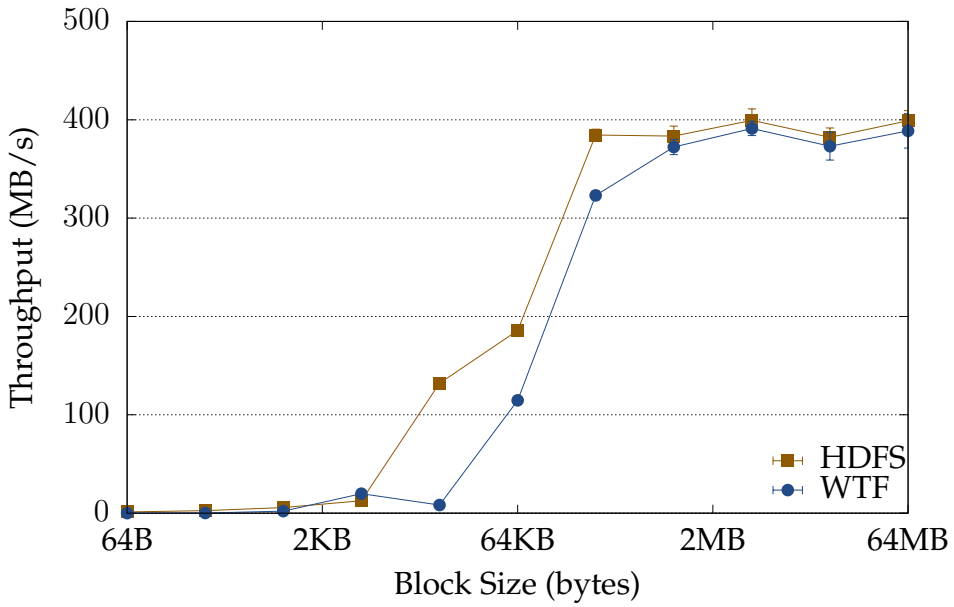


Figure 6.10: Aggregate throughput of a sequential write workload where writers make fixed size calls to "write". HDFS and WTF both provide applications with approximately $400\ \mathrm{MB/s}$ of goodput. Error bars report the standard error of the mean across seven trials.

tions. Figure 6.10 shows the results for block sizes between $64\ \mathrm{B}$ and $64\ \mathrm{MB}$. For writes greater than $1\ \mathrm{MB}$, WTF achieves 97% the throughput of HDFS. For $256\ \mathrm{kB}$ writes, WTF achieves 84% of the throughput of HDFS.

The latency for the two systems is similar, and directly correlated with the block size. Figure 6.11 shows the latency of writes across a variety of block sizes. We can see that WTF's median latency is very close to HDFS's median latency for larger writes, and that the 95th percentile latency for WTF is often lower than for HDFS.

Random Writes WTF enables applications to write at random offsets in a file without restriction. Because HDFS does not support random writes, we cannot use it as a baseline; instead, we will compare against the sequential write performance of WTF.

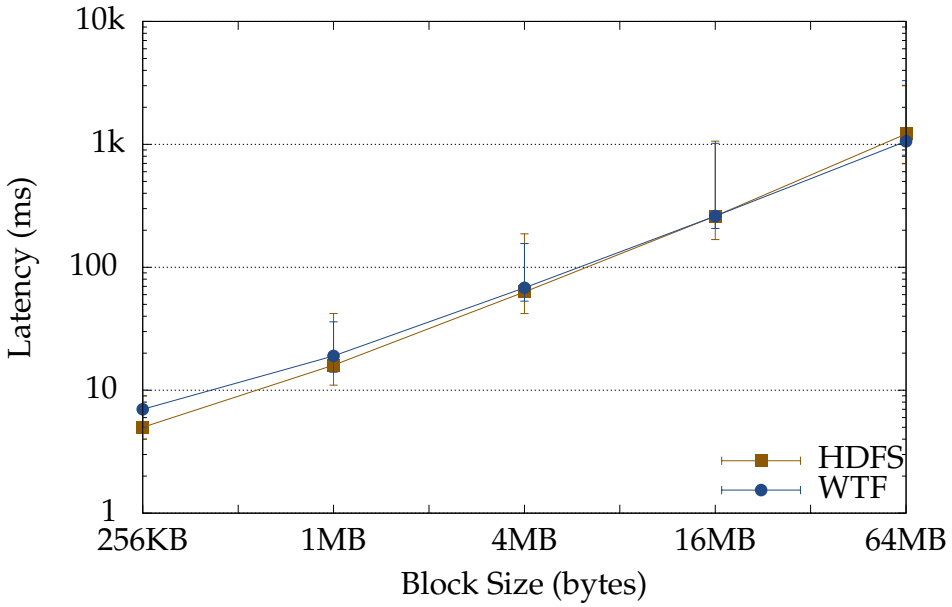


Figure 6.11: Median latency of write operations across a variety of write sizes. Error bars report the 5th and 95th percentile latencies.

Figure 6.12 shows the aggregate throughput achieved by clients writing to random offsets within WTF files. We see that the random write throughput is always within a factor of two of the sequential throughput, and that throughput converges as the size of the writes approaches 8 MB.

Because the common case for a sequential write and a random write in WTF differ only at the stage where metadata is written to HyperDex, we expect that such a difference in throughput is directly attributable to the metadata stage. HyperDex provides lower latency variance to applications with a small working set than applications with a large working set with no locality of access. We can see the difference this makes in the tail latency of WTF writes in Figure 6.13, which shows the median and 99th percentile latencies for both the sequential and random workloads. The median latency for both workloads is the same for all block sizes. For block sizes 4 MB and larger, the 99th percentile latencies are approximately the same as well. Writes less than 4 MB in size exhibit a signif-

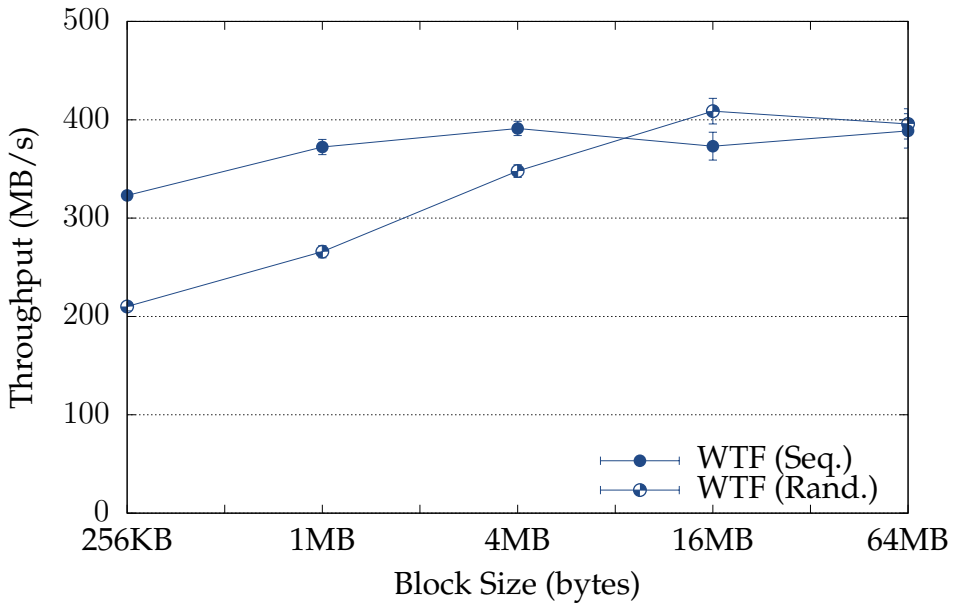


Figure 6.12: Aggregate throughput of concurrent writers making fixed size calls to "write" at random offsets within a file. Error bars report the standard error of the mean across seven trials.

icant difference in 99th percentile latency between the sequential and random workloads. These smaller writes spend more time updating HyperDex than writing to storage servers. We expect that further optimization of HyperDex would close the gap between sequential and random write performance.

Sequential Reads Batch processing applications often read large input files sequentially during both the map and reduce phases. Although a properly-written application will double-buffer to avoid small reads, applications are strictly better off if the filesystem does not rely on such behavior to enable high throughput. This experiment shows the extent to which WTF can be used by batch applications by reading through a file sequentially using a fixed-size buffer.

Figure 6.14 shows the aggregate throughput of concurrent readers reading through a 100 GB of data. We can see that for all read sizes, WTF's throughput

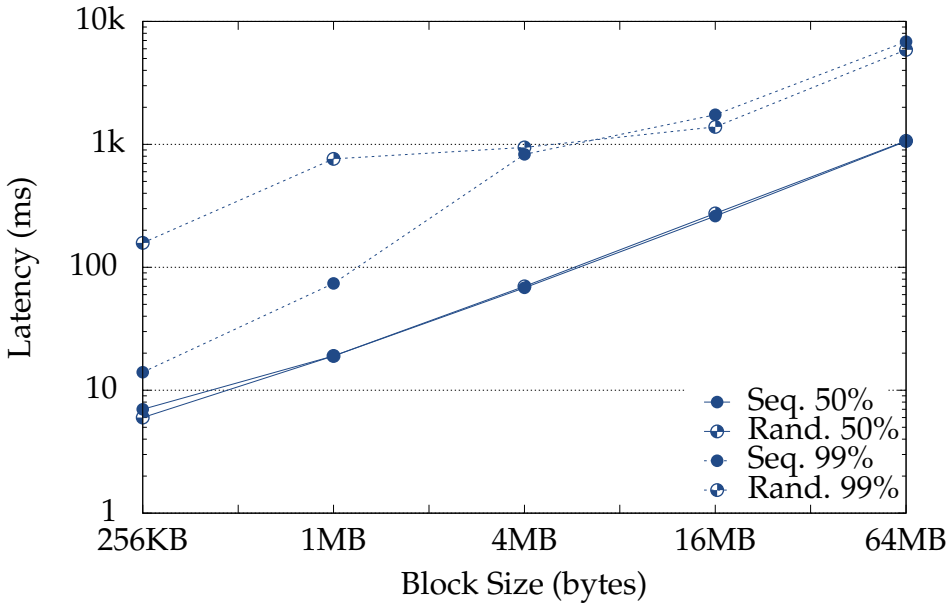


Figure 6.13: Median and 99th percentile latencies for sequential and random WTF writes. The median latency does not change between sequential and random write patterns.

is at least 80% the throughput of HDFS. The throughput reported here is double the throughput reported in the write benchmarks because only one of the two active replicas is consulted on each read. For smaller reads, WTF's throughput matches that of HDFS. The difference at larger sizes is largely an artifact of the implementations. HDFS uses readahead on both the clients and storage servers in order to improve throughput for streaming workloads. By default, the HDFS readahead is configured to be 4 MB, which is the point at which the systems start to exhibit different characteristics. Our preliminary WTF implementation does not have any readahead mechanism, and exhibits lower throughput.

Random Reads Applications built on a distributed filesystem, such as key-value stores or record-oriented applications often require random access to the files. Figure 6.15 shows the aggregate throughput of twelve concurrent random readers reading from randomly chosen offsets within 100 GB of data. We

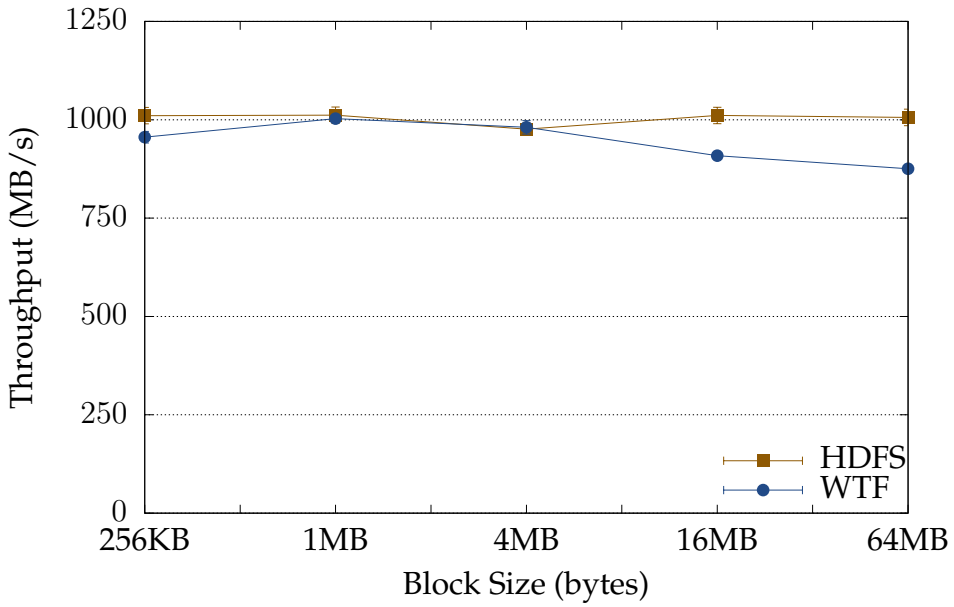


Figure 6.14: Aggregate throughput of concurrent readers reading fixed size blocks. HDFS and WTF both achieve approximately $900\ \mathrm{MB/s}$ of read throughput. Error bars report the standard error of the mean across seven trials.

can see that for reads of less than $16\ \mathrm{MB}$, WTF achieves significantly higher throughput—at its peak, WTF's throughput is $2.4\times$ the throughput of HDFS. Here, the readahead and client-side caching that helps HDFS with larger sequential read workloads adds overhead to HDFS that WTF does not incur. The 95th percentile latency of a WTF read is less than the median latency of a HDFS read for block sizes less than $4\ \mathrm{MB}$.

Scaling Workload This experiment varies the number of clients writing to the filesystem to explore how concurrency affects both latency and throughput. This benchmark employs the workload from the sequential-write benchmark with a $4\ \mathrm{MB}$ write size and a variable number of workload-generating clients.

Figures 6.16 and 6.17 shows the resulting throughput and latency for between one and twelve clients. We can see that the single client performance is approximately $60\ \mathrm{MB/s}$, while twelve clients sustain an aggregate throughput

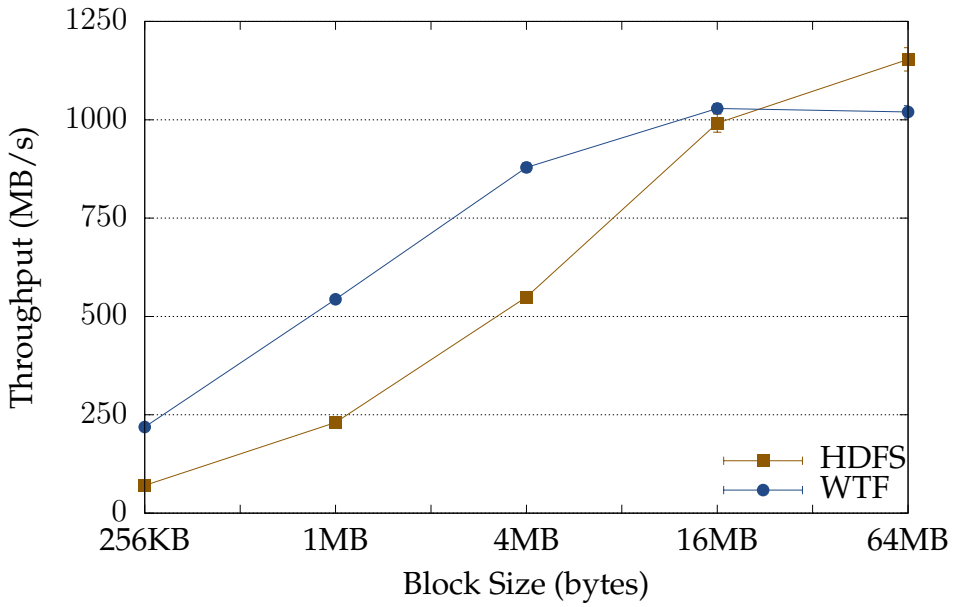


Figure 6.15: Aggregate throughput of random reads of varying size in a two-replicated deployment. We can see that WTF-backed applications achieve higher throughput than HDFS applications for a variety of small read sizes. Error bars indicate the standard error of the mean across seven trials.

of approximately $380\ \mathrm{MB/s}$. WTF's throughput is approximately the same as the throughput of HDFS for each data point. Running the same workload with forty-eight clients did not increase the throughput of either system beyond the throughput achieved with twelve clients, but did result in higher latency.

Fault Tolerance WTF's fault tolerance mechanism enables it to rapidly recover from failures. To demonstrate this mechanism, this benchmark performs sequential writes at a target throughput of $200\ \mathrm{MB/s}$. Figure 6.18 shows the throughput of the benchmark over time. Thirty seconds into the benchmark, one storage server is taken offline; ten seconds later, the coordinator reconfigures the system to remove the failed storage server. In the time between the failure and reconfiguration, clients may try to use the failed server, fail to write to it, and fall back to another server. This increased effort is reflected in the

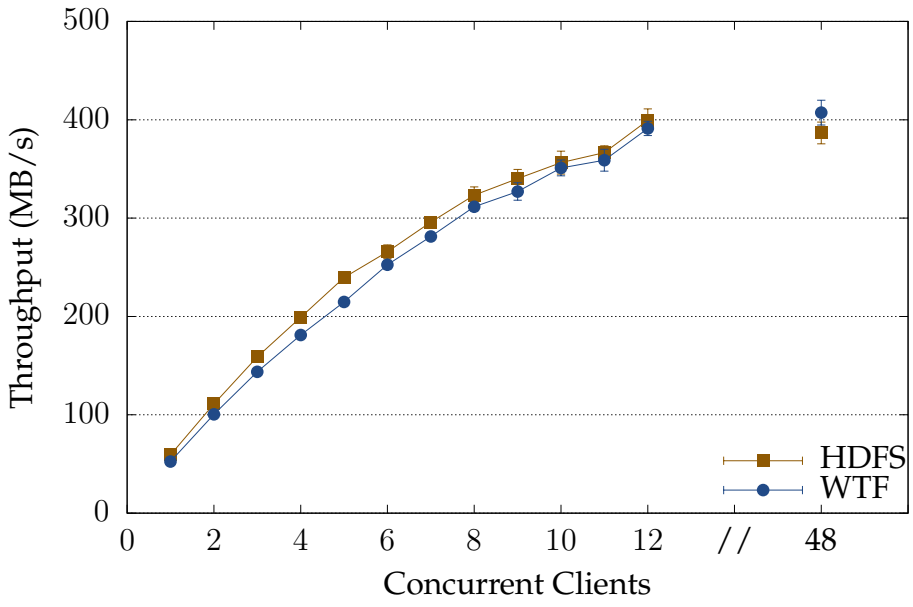


Figure 6.16: Aggregate throughput as the number of writers increases. Error bars show the standard error of the mean across seven trials.

lower throughput between failure and reconfiguration. After reconfiguration, throughput returns to to its rate before the failure. During the entire experiment, no writes failed, and the cluster as a whole remained available.

Garbage Collection This benchmark calculates the overhead of garbage collection on a storage server. As mentioned in Section 6.2.9, it is more efficient to collect files with more garbage than files with less garbage, and WTF preferentially garbage collects these larger files. Figure 6.19 shows the rate at which the cluster can collect garbage, for varying amounts of randomly located garbage, when all resources are dedicated to the task. We can see that when the cluster consists of 90% garbage, the cluster can reclaim this garbage at a rate of over $9\ \mathrm{GB}$ of garbage per second, because it need only write $1\ \mathrm{GB/s}$ to reclaim the garbage.

It is, however, impractical to dedicate all resources to garbage collection;

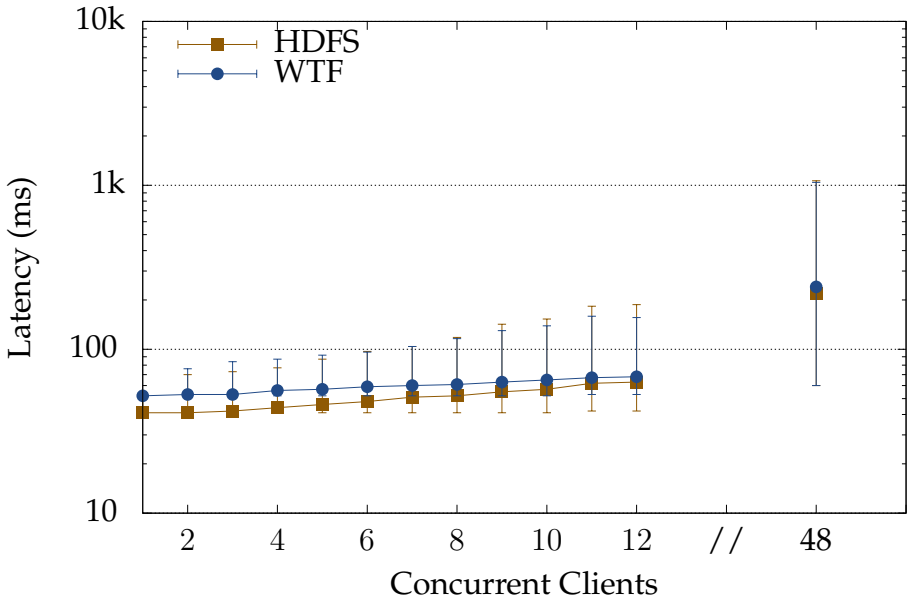


Figure 6.17: Median write latency as the number of writers increases. Error bars show the 5th and 95th percentile latencies.

instead, WTF dedicates only a fraction of I/O to the task. Storage servers initiate garbage collection when disk usage exceeds a configurable threshold, and ceases when the amount of garbage drops below 20%. Figure 6.19 shows that the maximum overhead required to maintain the system below this threshold is 4%.

Small Writes WTF's design is optimized for larger writes. The performance of smaller writes will largely be determined by the cost of updating the metadata. Writing a slice to the storage servers requires just one round trip because replicas are written to in parallel. Writing to the metadata store requires one round trip between client and the cluster, and multiple round trips within the cluster to propagate and commit the data. Further each write to the metdata requires writing approximately $50\ \mathrm{B}$ to HyperDex, so as writes to WTF shrink in size, the dominating cost becomes related to metadata.

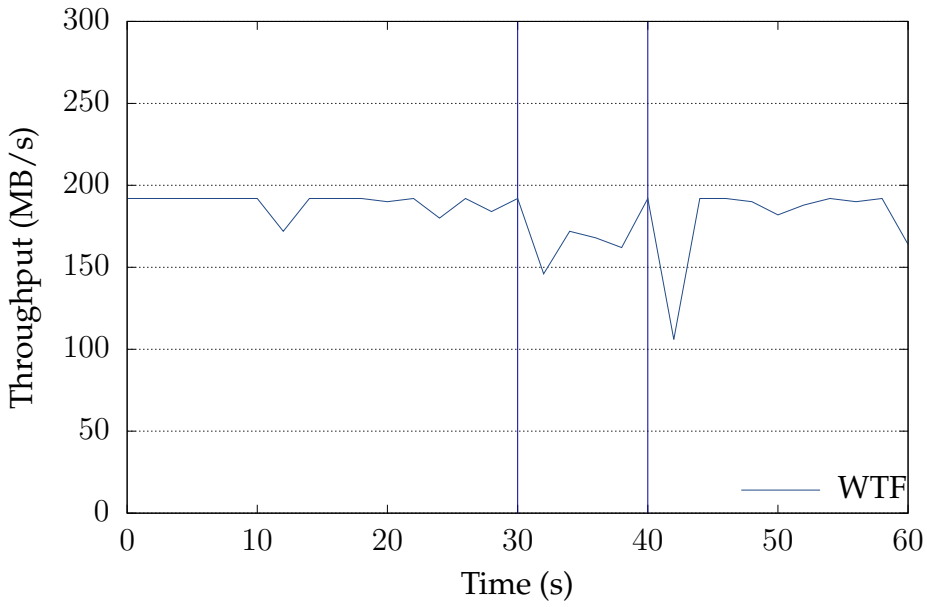


Figure 6.18: WTF tolerates failures—the failure occurs at the 30s mark—without a loss of availability and only a modest performance hit. Once the failure is corrected and the system reconfigured to explicitly ignore the failed server, the performance returns to normal.

Figure 6.20 focuses on a portion of the experiment shown in Figure 6.10, specifically writes less than 1 kB in size. HDFS achieves $140\times$ higher throughput for 64 B writes, while the difference is only a factor of $2.8\times$ for 1 kB writes. The figure also shows the calculated theoretical maximum throughput when the latency involved in writing to the metadata server is 2 ms, 5 ms, and 10 ms. This shows that the throughput of small operations is largely dependent upon the latency of metadata operations. Most workloads can avoid small operations with client side buffering, and further optimization of the metadata component could improve the throughput for small WTF writes.

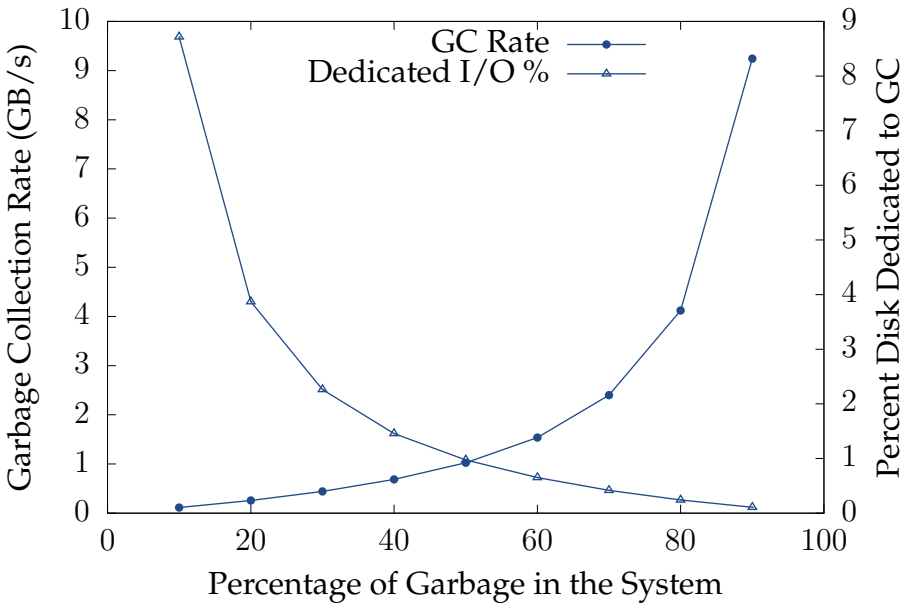


Figure 6.19: The maximum rate of garbage collection is positively correlated with the amount of garbage to be collected. Consequently, WTF dedicates a small fraction of its overall I/O to garbage collection.

6.5 Conclusion

This chapter described the Warp Transactional Filesystem (WTF), a new distributed filesystem that enables applications to operate on multiple files transactionally without requiring complex application logic. A new filesystem abstraction called *file slicing* further boosts performance by completely changing the filesystem interface to focus on metadata manipulation instead of data manipulation. Together, these features are a potent combination that enables a new class of high performance applications.

A broad evaluation shows that WTF achieves throughput and latency similar to industry-standard HDFS, while simultaneously offering stronger guarantees and a richer API. Sample applications show that WTF is usable in practice, and applications will often those built on a traditional filesystem—sometimes by orders of magnitude.

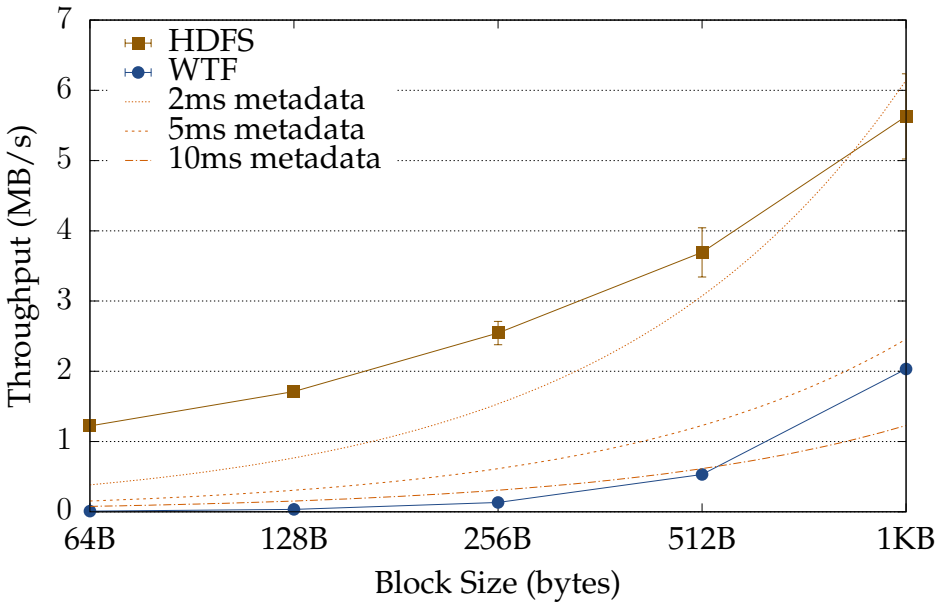


Figure 6.20: The time spent in metadata operations establishes an upper bound on the total throughput achievable by the system. This figure plots a portion of Figure 6.10 and theoretical maximum throughput for multiple metadata latencies.

CHAPTER 7

CONCLUSION

This thesis introduced four different search or transactional storage systems. This chapter looks at the broader impact of these systems, limitations of their design, and potential future spaces to explore. Of the four systems presented, HyperDex and Consus were released under open source licenses, Warp was released in binary form, and WTF was never released.

HyperDex was originally released in early 2012 as an open source project. The project saw lots of attention on social media for its fault tolerance and consistency guarantees, with many claiming it violated the CAP theorem [49]. The HyperDex team members publicly pushed back on the applicability of the CAP theorem to HyperDex: Specifically, the notion of a partition in CAP requires a system to tolerate practically any kind of failure and a node which receives a request must respond to the request without redirecting it elsewhere. HyperDex, in contrast, guarantees that operations will always remain consistent and will remain available so long as there are $f + 1$ replicas and at most f fail simultaneously.

This $f + 1$ fault tolerance guarantee is a core assumption of HyperDex, and by extension, Warp. HyperDex's implementation has no recourse for when all $f+1$ servers fail. Consequently, the HyperDex team often recommended people use $f = 2$ or $f = 3$ to ensure sufficient fault tolerance to overcome failure. This fault tolerance threshold, combined with the subspaces used to provide search, meant that a cluster deployment with s subspaces would have $s(f + 1)$, or one set of replicas for each subspace. A cluster with many subspaces could have more replica sets than exist replicas in each subspace.

It is this observation about HyperDex's fault tolerance that served as the in-

spiration for the Replex project [116]. Replex improves upon HyperDex's fault tolerance guarantees by allowing subspaces to remain fault tolerant when $f = 0$, as long as there are multiple subspaces. If a replica in an $f = 0$ subspace fails, Replex re-populates the subspace's replicas by retrieving the data from the other subspaces that exist. HyperDex provided a framework on which the Replex team "... added around 700 lines of code to HyperDex, around 500 of which were devoted to make data transfers during recovery performant." [116]. The changes made were largely to change the hash function used to map objects to servers, and to create the n:1 recovery mechanism for failures; the HyperDex value dependent chaining protocol required no modifications to naturally support Replex.

Replex also borrows from Warp's transaction chains to create a conditional indexing scheme. Like Warp, the protocol arranges the indices involved in a conditional operation—in Warp it would be a transaction and in Replex it would be a conditional insertion—into a single chain and propagates operations forward through the chain to collect votes about the validity of the operation. If and only if the operation reaches the end of the chain will it be deemed valid, and this information flows backwards through the chain to commit the outcome. Warp's key contributions are to describe in detail how to do this safely in the presence of concurrent transactions, while Replex omits a description of such concurrency control.

A more subtle consequence of HyperDex's fault tolerance guarantees, that HyperDex, Warp, and Replex all share is that they cannot endure a complete cluster outage. All of the systems assume that there will be at least $f + 1$ replicas and at most f will fail simultaneously. Consequently, the replication protocol inherently assumes that the entire cluster—or sufficiently large portions

thereof—cannot fail: Each link in a chain will remain available long enough to retain in-memory state for an in-flight operation. If multiple links fail, it violates the invariants that value-dependent chaining inherits from chain replication; the in-memory state may be made durable at later nodes in the chain while earlier nodes in the chain have completely forgotten the information because they suffered a failure and recovered quickly. Further, HyperDex and derivatives replace failed nodes in the chain by reconfiguring the cluster using a centralized entity backed by consensus. Reconfiguration can lead to periods of unavailability, and the way in which chains are reconfigured can cause a single failure to extend this unavailability as the chain rearranges non-failed nodes to uphold invariants. The results comparing HyperDex to Replex in the Replex work showcase this reconfiguration unavailability.

Work on Copysets [27] shows how to protect a cluster from the kinds of correlated failures to which HyperDex and its derivatives are vulnerable. In follow-up work [26], the HyperDex team worked with the authors of Copysets to improve the failure resilience of HyperDex with a dynamic algorithm expanding upon the Copysets work. This work makes it possible to tag replicas as being failure-independent. Each replica set will then include at least one replica from each failure domain. The tiered replication replica set selection algorithm presented in this work extends beyond HyperDex and could be applicable to any distributed system choosing replicas from a large pool of servers.

Consus avoids many of the pitfalls of HyperDex's fault tolerance scheme by taking a different assumption from HyperDex. Where HyperDex assumes that the system will experience at most f failures simultaneously and will remain available, it makes no assumption about what happens when the failure threshold is exceeded. Consus explicitly assumes that any number of servers

can fail simultaneously and upholds the guarantee that it can remain available if at most f fail. This difference is subtle, but important. The Consus guarantee covers strictly more behavior than the HyperDex guarantee and defines behavior for all cases likely to be encountered in practice.

Another change in Consus compared to HyperDex is a switch from using chain-oriented protocols to quorum-oriented protocols. While a chain-oriented protocol can offer theoretically higher throughput and may have acceptable latency, it is unavailable when any node fails until that node is removed or recovered. This complicates the system as it must have a mechanism by which nodes can be removed or recovered; further, the mechanism must balance accuracy and timeliness. Leaving a failed node in a chain for too long makes the chain unavailable for that period of time. Spuriously removing a node from a chain can make the chain less fault tolerant and introduce unnecessary churn when the node immediately decides to re-join the cluster. Balancing these two extremes is difficult to implement in practice, and has no single solution that's applicable to all environments. Because Consus uses quorum protocols, it can tolerate the absence of a node for extended periods of time without any unavailability. Qualitatively, this greatly simplifies the design and allows Consus to be more robust in more environments without manual tuning of the mechanisms for detecting and correcting failures—it could easily be a manual process diverted to a member of the operations team with a pager.

Consequently, Consus provides a stronger foundation on which to build a transactional storage system. Unfortunately, this comes at a cost: The design of Consus doesn't support or permit cheap "snapshot" reads; a scan of the entire database would be forced to lock each item in the database. This is somewhat inherent to its current design as the database state exists in the quorum of the

data centers, rather than in any one data center. Future work in this area could work to modify the Consus commit protocol to allow cheap, potentially lockfree reads like those present in Spanner.

While this thesis presents four different systems that are loosely tied together, the evolution of development and the relationship between the systems is as informational as the individual contributions of each system. Readers of this thesis should take note of the evolution of HyperDex, Warp, and Consus: The assumptions and functionality at the core of each is based upon the limitations or "wantings" experienced with the previous system, and their evolution over time unveils experience not present or known at the time of each individual work's publication.

BIBLIOGRAPHY

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, Available, And Reliable Storage For An Incompletely Trusted Environment. In Proceedings of the *Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[2] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In Proceedings of the *SIGMOD International Conference on Management of Data*, pages 23-34, San Jose, California, May 1995.

[3] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A Practical Scalable Distributed B-Tree. In *Proceedings of the VLDB Endowment*, 1(1):598-609, 2008.

[4] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A New Paradigm For Building Scalable Distributed Systems. In Proceedings of the *Symposium on Operating Systems Principles*, pages 159-174, Stevenson, Washington, October 2007.

[5] Deniz Altınbüken and Emin Gün Sirer. Commodifying Replicated State Machines With OpenReplica. Cornell University, Technical Report, 2012.

[6] Amazon Web Services. Elastic Block Store. http://aws.amazon.com/ebs/.

[7] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array Of Wimpy Nodes. In Proceedings of the *Symposium on Operating Systems Principles,* pages 1-14, Big Sky, Montana, October 2009.

[8] Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In Proceedings of the *Symposium on Operating Systems Principles,* pages 109-126, Copper Mountain, Colorado, December 1995.

[9] Apache Hadoop. http://hadoop.apache.org/.

[10] Apache Hadoop Jira. DFS Used Space Is Not Correct Computed On Frequent Append Operations. https://issues.apache.org/jira/browse/HDFS-6489.

[11] Apache Hadoop Jira. Increase The Default Block Size. https://issues.apache.org/jira/browse/HDFS-4053.

[12] Apache HBase. http://hbase.apache.org/.

[13] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination-Avoiding Database Systems. In *CoRR,* abs/1402.2237, 2014.

[14] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage For Interactive Services. In Proceedings of the *Conference on Innovative Data Systems Research,* pages 223-234, Asilomar, California, January 2011.

[15] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, David Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony I. T. Rowstron. Pelican: A Building Block For Exascale Cold Data Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 351-365, Broomfield, Colorado, October 2014.

[16] Rudolf Bayer. The Universal B-Tree For Multidimensional Indexing: General Concepts. In Proceedings of the *Worldwide Computing and its Applications,* pages 198-209, Tsukuba, Japan, March 1997.

[17] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding A Needle In Haystack: Facebook's Photo Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 47-60, Vancouver, Canada, October 2010.

[18] Richard Ernest Bellman. Dynamic Programming. Princeton University Press, Princeton, New Jersey, 1957.

[19] Jon Louis Bentley. Multidimensional Binary Search Trees Used For Associative Searching. In *Communications of the ACM,* 18(9):509-517, 1975.

[20] Philip A. Bernstein and Nathan Goodman. Concurrency Control In Distributed Database Systems. In *ACM Computing Surveys,* 13(2):185-221, 1981.

[21] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In Proceedings of the *SIGCOMM Conference,* pages 353-366, Portland, Oregon, August 2004.

[22] Michael Burrows. The Chubby Lock Service For Loosely-Coupled Distributed Systems. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 335-350, Seattle, Washington, November 2006.

[23] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using Distributed Disk Striping To Provide High I/O Data Rates. In *Computing Systems*, 4(4):405-436, 1991.

[24] Min Cai, Martin R. Frank, Jinbo Chen, and Pedro A. Szekely. MAAN: A Multi-Attribute Addressable Network For Grid Information Services. In Proceedings of the *International Workshop on Grid Computing*, pages 184-191, Phoenix, Arizona, November 2003.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. BigTable: A Distributed Storage System For Structured Data. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 205-218, Seattle, Washington, November 2006.

[26] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gün Sirer. Tiered Replication: A Cost-Effective Alternative To Full Cluster Geo-Replication. In Proceedings of the *USENIX Annual Technical Conference*, Santa Clara, California, July 2015.

[27] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John K. Ousterhout, and Mendel Rosenblum. Copysets: Reducing The Frequency Of Data Loss In Cloud Storage. In Proceedings of the *USENIX Annual Technical Conference*, San Jose, California, June 2013.

[28] Austin T. Clements, Dan R. K. Ports, and David R. Karger. Arpeggio: Metadata Searching And Content Sharing With Chord. In Proceedings of the *International Workshop on Peer-to-Peer Systems,* pages 58-68, La Jolla, California, February 2005.

[29] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment,* 1(2):1277-1288, 2008.

[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems With YCSB. In Proceedings of the *Symposium on Cloud Computing,* pages 143-154, Indianapolis, Indiana, June 2010.

[31] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. In *ACM Transactions on Computer Systems,* 31(3):8, 2013.

[32] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito,

Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 261-264, Hollywood, California, October 2012.

[33] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In Proceedings of the *USENIX Annual Technical Conference*, 2012.

[34] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In Proceedings of the *International Workshop on the Web and Databases*, pages 25-30, Paris, France, June 2004.

[35] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage With CFS. In Proceedings of the *Symposium on Operating Systems Principles*, pages 202-215, Banff, Canada, October 2001.

[36] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. In *Communications of the ACM*, 53(1):72-77, 2010.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In Proceedings of the *Symposium on Operating Systems Principles*, pages 205-220, Stevenson, Washington, October 2007.

[38] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In Proceedings of the *Symposium on*

Networked System Design and Implementation, pages 401-414, Seattle, Washington, April 2014.

[39] Robert Escriva and Emin Gün Sirer. The Design And Implementation Of The Warp Transactional Filesystem. In Proceedings of the *Symposium on Networked System Design and Implementation*, Santa Clara, California, March 2016.

[40] Robert Escriva and Robbert van Renesse. Consus: Taming The Paxi. In *CoRR*, abs/1612.03457, 2016.

[41] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proceedings of the *SIGCOMM Conference*, pages 25-36, Helsinki, Finland, August 2012.

[42] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight Multi-Key Transactions For Key-Value Stores. In *CoRR*, abs/1509.07815, 2015.

[43] Daniel Gómez Ferro, Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-Free Transactional Support For Distributed Data Stores. Poster Session. Symposium on Operating Systems Principles, Cascais, Portugal, 2011.

[44] David Gale and Lloyd S. Shapley. College Admissions And The Stability Of Marriage. In *The American Mathematical Monthly*, 69(1):9-15, 1962.

[45] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One Torus To Rule Them All: Multidimensional Queries In P2P Systems. In Proceedings of the *International Workshop on the Web and Databases*, pages 19-24, Paris, France, June 2004.

[46] João Garcia, Paulo Ferreira, and Paulo Guedes. The PerDiS FS: A Transactional File System For A Distributed Persistent Store. In Proceedings of the *European SIGOPS Workshop*, pages 189-194, Sintra, Portugal, September 1998.

[47] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In Proceedings of the *Symposium on Operating Systems Principles*, pages 29-43, Bolton Landing, New York, October 2003.

[48] Garth A. Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems*, pages 92-103, San Jose, California, October 1998.

[49] Seth Gilbert and Nancy A. Lynch. Brewer's Conjecture And The Feasibility Of Consistent, Available, Partition-Tolerant Web Services. In *SIGACT News*, 33(2):51–59, 2002.

[50] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas E. Anderson. Scalable Consistency In Scatter. In Proceedings of the *Symposium on Operating Systems Principles*, pages 15-28, Cascais, Portugal, October 2011.

[51] Jim Gray. A Transaction Model. In *Automata, Languages and Programming*, 85:282-298, 1980.

[52] Jim Gray and Leslie Lamport. Consensus On Transaction Commit. In *ACM Transactions on Database Systems*, 31(1):133-160, 2006.

[53] Steven D. Gribble. A Design Framework And A Scalable Storage Platform To Simplify Internet Service Construction. U.C. Berkeley, 2000.

[54] Antonin Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In Proceedings of the *SIGMOD International Conference on Management of Data,* pages 47-57, Boston, Massachusetts, June 1984.

[55] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *ACM Transactions on Computer Systems,* 13(3):274-310, 1995.

[56] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition For Concurrent Objects. In *ACM Transactions on Programming Languages and Systems,* 12(3):463-492, 1990.

[57] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale And Performance In A Distributed File System. In *ACM Transactions on Computer Systems,* 6(1):51-81, 1988.

[58] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination For Internet-Scale Systems. In Proceedings of the *USENIX Annual Technical Conference,* 2010.

[59] Hypertable. http://http://hypertable.org/.

[60] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing And Random Trees: Distributed Caching Protocols For Relieving Hot Spots On The World Wide Web. In Proceedings of the *ACM Symposium on Theory of Computing,* pages 654-663, El Paso, Texas, May 1997.

[61] Farnoush Banaei Kashani and Cyrus Shahabi. SWAM: A Family Of Access Methods For Similarity-Search In Peer-To-Peer Data Networks. In Proceedings of the *International Conference on Information and Knowledge Management,* pages 304-313, Washington, D.C., November 2004.

[62] Allen Klinger. Patterns And Search Statistics. In *Optimizing Methods in Statistics,* pages 303-337, 1971.

[63] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-Data Center Consistency. In Proceedings of the *European Conference on Computer Systems,* pages 113-126, Prague, Czech Republic, April 2013.

[64] H. T. Kung and John T. Robinson. On Optimistic Methods For Concurrency Control. In *ACM Transactions on Database Systems,* 6(2):213-226, 1981.

[65] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. In Proceedings of the *International Workshop on Large Scale Distributed Systems and Middleware,* Big Sky, Montana, October 2009.

[66] Leslie Lamport. Generalized Consensus And Paxos. Microsoft Research, Technical Report MSR-TR-2005-33, 2005.

[67] Leslie Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems,* 16(2):133-169, 1998.

[68] Leslie Lamport. Time, Clocks, And The Ordering Of Events In A Distributed System. In *Communications of the ACM,* 21(7):558-565, 1978.

[69] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In Proceedings of the *Architectural Support for Programming*

Languages and Operating Systems, pages 84-92, Cambridge, Massachusetts, October 1996.

[70] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency Without Concurrency Control. In *The Computing Research Repository*, abs/0907.0929, 2009.

[71] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In Proceedings of the *Symposium on Operating Systems Principles*, pages 1-13, Cascais, Portugal, October 2011.

[72] Linux Kernel Developers. Documentation For /proc/sys/vm/*. https://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[73] Barbara Liskov and Rodrigo Rodrigues. Transactional File Systems Can Be Fast. In Proceedings of the *Proceedings of the 11st ACM SIGOPS European Workshop, Leuven, Belgium, September 19-22, 2004*, pages 5-11, 2004.

[74] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle For Eventual: Scalable Causal Consistency For Wide-Area Storage With COPS. In Proceedings of the *Symposium on Operating Systems Principles*, pages 401-416, Cascais, Portugal, October 2011.

[75] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics For Low-Latency Geo-Replicated Storage. In Proceedings of the *Symposium on Networked System Design and Implementation*, pages 313-328, Lombard, Illinois, April 2013.

[76] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Con-

sistency: Measuring And Understanding Consistency At Facebook. In Proceedings of the *Symposium on Operating Systems Principles,* Monterey, California, October 2015.

[77] Hatem A. Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-Latency Multi-Datacenter Databases Using Replicated Commit. In *Proceedings of the VLDB Endowment,* 6(9):661-672, 2013.

[78] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness For Fast Multicore Key-Value Storage. In Proceedings of the *European Conference on Computer Systems,* pages 183-196, Bern, Switzerland, April 2012.

[79] Kirk McKusick and Sean Quinlan. GFS: Evolution On Fast-Forward. In *Communications of the ACM,* 53(3):42-49, 2010.

[80] Memcached. http://memcached.org/.

[81] James W. Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-Scale Block Storage For Cloud-Oblivious Applications. In Proceedings of the *Symposium on Networked System Design and Implementation,* pages 257-273, Seattle, Washington, April 2014.

[82] MongoDB. http://www.mongodb.org/.

[83] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There Is More Consensus In Egalitarian Parliaments. In Proceedings of the *Symposium on Operating Systems Principles,* pages 358-372, 2013.

[84] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency From Distributed Transactions. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 479-494, Broomfield, Colorado, October 2014.

[85] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control And Consensus For Commits Under Conflicts. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 517-532, Savannah, Georgia, November 2016.

[86] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. In *ACM Transactions on Database Systems*, 9(1):38-71, 1984.

[87] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen S. Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 1-15, Hollywood, California, October 2012.

[88] Michael A. Olson. The Design And Implementation Of The Inversion File System. In Proceedings of the *USENIX Winter Technical Conference*, pages 205-218, San Diego, California, January 1993.

[89] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast Crash Recovery In RAMCloud. In Proceedings of the *Symposium on Operating Systems Principles*, pages 29-41, Cascais, Portugal, October 2011.

[90] Jack A. Orenstein and T. H. Merrett. A Class Of Data Structures For Asso-

ciative Searching. In Proceedings of the *Symposium on Principles of Database Systems*, pages 181-190, Waterloo, Canada, April 1984.

[91] Daniel Peng and Frank Dabek. Large-Scale Incremental Processing Using Distributed Transactions And Notifications. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 251-264, Vancouver, Canada, October 2010.

[92] Larry Peterson, Andy Bavier, and Sapan Bhatia. VICCI: A Programmable Cloud-Computing Research Testbed. Princeton, Technical Report TR-912-11, 2011.

[93] Project Voldemort. http://project-voldemort.com/.

[94] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos To Build A Scalable, Consistent, And Highly Available Datastore. In *Proceedings of the VLDB Endowment*, 4(4):243-254, 2011.

[95] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of the *SIGCOMM Conference*, pages 161-172, San Diego, California, August 2001.

[96] Riak. http://basho.com/.

[97] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, And Routing For Large-Scale Peer-To-Peer Systems. In Proceedings of the *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329-350, 2001.

[98] Hanan Samet. Spatial Data Structures. In *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 361-385, 1995.

[99] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable Transactional P2P Key/value Store. In Proceedings of the *SIGPLAN Workshop on ERLANG*, pages 41-48, Victoria, Canada, 2008.

[100] Richard D. Schlichting and Fred B. Schneider. Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems. In *ACM Transactions on Computer Systems*, 1(3):222-238, 1983.

[101] Charles Schmidt and Manish Parashar. Enabling Flexible Queries With Guarantees In P2P Systems. In *Internet Computing Journal*, pages 19-26, 2004.

[102] Frank B. Schmuck and Roger L. Haskin. GPFS: A Shared-Disk File System For Large Computing Clusters. In Proceedings of the *Conference on File and Storage Technologies*, pages 231-244, Monterey, California, January 2002.

[103] Frank B. Schmuck and James C. Wyllie. Experience With Transactions In QuickSilver. In Proceedings of the *Symposium on Operating Systems Principles*, pages 239-253, Pacific Grove, California, October 1991.

[104] Daniele Sciascia and Fernando Pedone. Geo-Replicated Storage With Scalable Deferred Update Replication. In Proceedings of the *International Conference on Dependable Systems and Networks*, pages 1-12, Budapest, Hungary, June 2013.

[105] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable Deferred Update Replication. In Proceedings of the *International Conference on Dependable Systems and Networks*, pages 1-12, Boston, Massachusetts, June 2012.

[106] Seagate Technology LLC. Lustre Filesystem. http://lustre.org/.

[107] Margo I. Seltzer. Transaction Support In A Log-Structured File System. In Proceedings of the *IEEE International Conference on Data Engineering*, pages 503-510, Vienna, Austria, April 1993.

[108] Yanfeng Shu, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Supporting Multi-Dimensional Range Queries In Peer-To-Peer Systems. In Proceedings of the *IEEE International Conference on Peer-to-Peer Computing*, pages 173-180, Konstanz, Germany, August 2005.

[109] Dale Skeen and Michael Stonebraker. A Formal Model Of Crash Recovery In A Distributed System. In *IEEE Transactions on Software Engineering*, 9(3):219-228, 1983.

[110] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage For Geo-Replicated Systems. In Proceedings of the *Symposium on Operating Systems Principles*, pages 385-400, Cascais, Portugal, October 2011.

[111] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access Via Lightweight Kernel Extensions. In Proceedings of the *Conference on File and Storage Technologies*, pages 29-42, San Francisco, California, February 2009.

[112] Richard P. Spillane, Pradeep J. Shetty, Erez Zadok, Sagar Dixit, and Shrikar Archak. An Efficient Multi-Tier Tablet Server Storage Architecture. In Proceedings of the *Symposium on Cloud Computing*, pages 1-14, Cascais, Portugal, October 2011.

[113] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service For In-

ternet Applications. In Proceedings of the *SIGCOMM Conference*, pages 149-160, San Diego, California, August 2001.

[114] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End Of An Architectural Era (It's Time For A Complete Rewrite). In Proceedings of the *International Conference on Very Large Data Bases*, pages 1150-1160, 2007.

[115] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. F4: Facebook's Warm BLOB Storage System. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 383-398, Broomfield, Colorado, October 2014.

[116] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. Replex: A Scalable, Highly Available Multi-Index Data Store. In Proceedings of the *USENIX Annual Technical Conference*, pages 337–350, Denver, Colorado, June 2016.

[117] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In Proceedings of the *Symposium on Operating Systems Principles*, pages 224-237, Saint Malo, France, October 1997.

[118] Härder Theo. Observations On Optimistic Concurrency Control Schemes. In *Information Systems*, 9(2):111-120, 1984.

[119] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN Replication And Scalable Metadata Management For Distributed File Sys-

tems. In Proceedings of the *Conference on File and Storage Technologies*, pages 1-14, Santa Clara, California, February 2015.

[120] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions For Partitioned Database Systems. In Proceedings of the *SIGMOD International Conference on Management of Data*, pages 1-12, Scottsdale, Arizona, May 2012.

[121] Robbert van Renesse and Fred B. Schneider. Chain Replication For Supporting High Throughput And Availability. In Proceedings of the *Symposium on Operating System Design and Implementation*, pages 91-104, San Francisco, California, December 2004.

[122] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness In The Salus Scalable Block Store. In Proceedings of the *Symposium on Networked System Design and Implementation*, pages 357-370, Lombard, Illinois, April 2013.

[123] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance Of The Panasas Parallel File System. In Proceedings of the *Conference on File and Storage Technologies*, pages 17-33, San Jose, California, February 2008.

[124] Charles P. Wright, Richard P. Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics To The File System. In *ACM Transactions on Storage*, 3(2), 2007.

[125] Chi Zhang, Arvind Krishnamurthy, and Randolph Y. Wang. SkipIndex: Towards A Scalable Peer-To-Peer Index Service For High Dimensional Data. Princeton, Technical Report TR-703-04, 2004.

[126] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability With Low Latency In Geo-Distributed Storage Systems. In Proceedings of the *Symposium on Operating Systems Principles*, pages 276-291, 2013.

[127] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: A Fault-Tolerant Wide-Area Application Infrastructure. In *SIGCOMM Computer Communications Review*, 32(1):81, 2002.